ESCUELA SUPERIOR DE INGENIERÍA

INGENIERÍA INFORMÁTICA

REINGENIERÍA DE UN GENERADOR DE CASOS DE PRUEBA POR ALGORITMOS GENÉTICOS PARA WS-BPEL 2.0

Álvaro Cortijo García

11 de septiembre de 2015

ESCUELA SUPERIOR DE INGENIERÍA

INGENIERÍA INFORMÁTICA

REINGENIERÍA DE UN GENERADOR DE CASOS DE PRUEBA POR ALGORITMOS GENÉTICOS PARA WS-BPEL 2.0

- Departamento: Departamento de Ingeniería Informática
- Directores del proyecto: Antonia Estero Botaro y Antonio García Domínguez
- Autor del proyecto: Álvaro Cortijo García

Cádiz, 11 de septiembre de 2015

Fdo.: Álvaro Cortijo García

Agradecimientos

- A mis padres, por el apoyo incondicional en todo momento.
- A los profesores del grupo UCASE que me han permitido colaborar con ellos y realizar este proyecto.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Motivación y contexto

En este capítulo del documento se detalla la motivación del proyecto mediante una introducción en la que se se comentan los principales temas en los que se basa este proyecto fin de carrera. Además, se describen los objetivos que se tratan de satisfacer en el mismo. A lo largo de este capítulo también se detallan una serie de conceptos básicos, como la prueba de mutaciones, los algorítmos genéticos y los distintos lenguajes y tecnologías que han sido necesarias para el desarrollo del proyecto.

1.1. Introducción

El grupo de investigación UCASE de la Universidad de Cádiz trabaja en varias líneas de investigación, entre las cuales se encuentran:

- La aplicación de la técnica de prueba de mutaciones al lenguaje WS-BPEL 2.0 (Web Services Business Process Execution Language) [1].
- La generación automática de casos de prueba para composiciones WS-BPEL 2.0.

El lenguaje WS-BPEL 2.0 permite crear procesos de negocio a partir de servicios web preexistentes, de manera que facilita la composición de dichos servicios web respecto a lenguajes tradicionales. Debido al creciente uso que los servicios web están experimentando, la prueba de este tipo de software toma una gran importancia.

Desde hace varios años, el grupo de investigación UCASE lleva trabajando en la línea de investigación consistente en la aplicación de la técnica de prueba de mutaciones a composiciones WS-BPEL 2.0. En este sentido, ha definido e implementado un conjunto de operadores de mutación para WS-BPEL 2.0, estos operadores permiten modelar los errores que suelen cometer los programadores o aplicar criterios de cobertura. Mediante la prueba de mutaciones podemos medir la calidad del conjunto de casos de pruebas empleado.

Diversos autores han propuesto la aplicación de técnicas de búsqueda evolutiva, en general, y algoritmos genéticos [2] en particular, a la generación de casos de prueba. El amplio uso que se observa en la literatura de los algoritmos genéticos para la generación de casos de prueba y los buenos resultados que suelen obtener han hecho que el grupo UCASE se decante por estos para basar en ellos su herramienta Rodan [3] de generación de casos de prueba para composiciones WS-BPEL. El objetivo principal que se persigue con esta herramienta es la generación de casos de prueba que maten el mayor número posible de los mutantes producidos a partir de una composición WS-BPEL.

No obstante, los resultados producidos por la herramienta hasta la fecha no han sido satisfactorios, por lo que es necesario reevaluar su diseño y comprobar que se está implementando correctamente el algoritmo genético diseñado por el grupo UCASE. En este proyecto se mejorará el diseño de la herramienta y se validará su implementación interna contra el algoritmo diseñado, mejorándolo en lo

necesario de acuerdo a los experimentos que se realicen sobre las composiciones WS-BPEL del grupo UCASE. El objetivo final de este proyecto será la mejora de los resultados científicos producidos por Rodan.

1.2. Objetivos

El objetivo principal del proyecto consiste en la reingeniería de un sistema de generación de casos de prueba basado en un algoritmo genético, de tal manera que no presente los problemas actuales, esté bien definido y sea útil para alcanzar resultados de investigación significativos.

Los individuos de la población que empleará el algoritmo genético son casos de prueba y el objetivo de dicho algoritmo genético es la generación, especialmente a través de cruces y mutaciones, de nuevos casos de prueba que maten al mayor número de mutantes producidos a partir de una composición WS-BPEL 2.0.

También se realizarán estudios de investigación aplicando la herramienta a determinadas composiciones WS-BPEL 2.0 y evaluando la efectividad de los conjuntos de casos de prueba producidos por la herramienta y la eficiencia del proceso implementado.

Así mismo, se mejorará la visualización de los resultados presentados por la herramienta, proporcionando la generación de forma automática de un informe en formato *html* que contenga información sobre las estadísticas y los resultados obtenidos en una ejecución del algoritmo genético.

1.3. Conceptos básicos

Esta sección describe la técnica de la prueba de mutaciones y las tecnologías que se han manejado para la realización de este proyecto.

1.3.1. La prueba de mutaciones

La *prueba de mutaciones* [4] es una técnica de prueba del software basada en fallos. La prueba de mutaciones consiste en introducir pequeños cambios sintácticos en el programa a probar mediante la aplicación de los denominados *operadores de mutación*. Los operadores de mutación suelen modelar los principales fallos que suelen cometer los programadores al implementar un programa o bien, pueden permitir aplicar criterios de cobertura. Los nuevos programas generados mediante la aplicación de los operadores de mutación se denominan *mutantes*. La prueba de mutaciones permite conseguir dos objetivos: probar el programa y medir la calidad del conjunto de casos de pruebas que se está utilizando, permitiendo además mejorar su calidad si no se considera adecuada.

A continuación, podemos observar un ejemplo de aplicación de un operador de mutación. El código original es una expresión lógica en la que comprobamos si la variable *stock* es mayor que una determinada cantidad.

```
1 $stock > 100
```

El mutante generado, a partir de ese código, por un operador de mutación podría ser el de cambiar el operador lógico por otro, y en vez de comprobar si la variable *stock* es mayor que una determinada cantidad, comprobar si es menor.

```
1 $stock < 100
```

Una vez que se han obtenido los distintos mutantes de un determinado programa, se ejecutan contra un conjunto de casos de prueba y se comparan los resultados obtenidos por los mutantes con los del

programa original. Si la salida del mutante coincide con la del programa original, decimos que el mutante ha quedado vivo. Por contra, si ambas salidas no coinciden decimos que el mutante ha muerto. También existe la posibilidad de que un mutante no pueda ser ejecutado, en este caso decimos que el mutante es no válido.

Un mutante puede quedar vivo por dos razones:

- No disponemos del caso de prueba adecuado que lo detecte, decimos que se trata de un mutante persistente.
- Se trata de un mutante equivalente, es decir, un mutante que siempre va a producir la misma salida que el programa original.

En el primer caso, añadiendo nuevos casos de prueba a nuestro conjunto podremos matar al mutante, aumentando así la calidad del conjunto de casos de prueba. Cuando un conjunto de casos de prueba mata a todos los mutantes no equivalentes producidos por un programa se dice que es un conjunto de casos de prueba *adecuado*.

La calidad de un conjunto de casos de prueba se puede calcular mediante el cociente entre el número de mutantes muertos y el número de mutantes no equivalentes.

1.3.2. Algoritmos genéticos

Los algoritmos genéticos son técnicas de búsqueda basadas en los mecanismos de evolución biológica y selección natural. A partir de una determinada población de soluciones llamadas individuos, mejoran a dicha población a través de métodos aleatorios similares a los que intervienen en la evolución de las especies: la recombinación genética y la mutación. Así mismo, a través de distintas generaciones los algoritmos genéticos realizan un proceso de selección de individuos de acuerdo a un determinado criterio que se corresponde con la función de aptitud empleada. De esta manera mejora a la población puesto que los individuos con mayor aptitud son los que tienen mayor probabilidad de ser seleccionados para constituir los individuos de la siguiente generación a partir del cruce y la mutación de la información de dichos individuos, al igual que ocurre en la evolución de las especies mediante la selección natural, donde los individuos mejor adaptados son los que sobreviven y transmiten su material genético. En este sentido, los algoritmos genéticos son muy apropiados para resolver problemas de búsqueda y optimización.

La función de aptitud mide la calidad de cada individuo que conforma la población, representando cada individuo a una solución del problema concreto a resolver. La elección tanto de la función de aptitud como de la forma de codificar los individuos dependen considerablemente del problema que se está estudiando. Por otro lado, al tener más opciones de ser elegidos los individuos con mayor aptitud, a lo largo de las distintas generaciones que el algoritmo genético realiza, la aptitud promedio de la población mejora progresivamente.

Los dos tipos de operadores que emplean los algoritmos genéticos son los operadores de selección y reproducción. Los operadores de selección se encargan de elegir los individuos de la población de una determinada generación para participar en la reproducción, esta selección se realiza en relación a la aptitud de los individuos, por lo que cuanto mayor sea la aptitud de un determinado individuo mayor será la probabilidad de que dicho individuo sea seleccionado. Por su parte, los operadores de reproducción son los encargados de introducir la diversidad genética en la población mediante el intercambio de información y modificaciones de ésta, de manera que introducen nuevos individuos en una población a través de operaciones sobre los individuos de la población de la generación anterior. Hay dos tipos de operadores de reproducción: Los operadores de cruce y los operadores de mutación.

El objetivo del operador de cruce es obtener dos nuevos individuos a partir de una pareja de individuos de la población de la generación anterior. La realización de este cruce tiene lugar mediante una recombinación de la información genética de los padres, dando lugar a sus dos descendientes.

Con los operadores de mutación se modifica la información de un individuo para producir otro nuevo. De esta manera, se modifica el valor de algunos de los componentes del individuo y se consigue alcanzar zonas del espacio de búsqueda que podrían no ser alcanzadas por los individuos ya existentes en la población ni por los otros tipos de operadores de reproducción.

Además, el algoritmo genético para de realizar generaciones cuando alcanza una determinada condición de terminación, como puede ser un número máximo de generaciones establecido o que la aptitud máximo o promedia de la población no presente mejoras durante un determinado número de generaciones.

1.3.3. El lenguaje WS-BPEL 2.0

El lenguaje WS-BPEL 2.0 (Web Services Business Process Execution Language) es un lenguaje basado en XML estandarizado por OASIS [1] con el que podemos especificar el comportamiento de un proceso de negocio, basado en la interacción con servicios web. Estos servicios se especifican mediante el lenguaje WSDL (Web Services Description Language), a través de este lenguaje se pueden definir los mensajes mediante los que pueden interactuar los distintos servicios web durante su comunicación, así como, la forma de establecer la comunicación entre los distintos servicios web y los protocolos de comunicación usados.

La estructura de un proceso WS-BPEL se divide en las siguientes cuatro secciones:

1. Definición de las relaciones con los socios externos, es decir, el cliente que utiliza el proceso de negocio y los WS a los que llama el proceso.
2. Definición de las variables que emplea el proceso, mediante Web Services Description Language (WSDL) y XML Schema (XSD).
3. Definición de los distintos tipos de manejadores que puede utilizar el proceso. Los distintos manejadores disponibles en WS-BPEL son:
   - **Manejadores de fallos:** A través de los elementos *catch* permiten determinar las actividades a llevar a cabo en caso de que se produzca un fallo específico, y mediante el elemento *catchall* las actividades relacionadas con cualquier otro fallo no especificado anteriormente.
   - **Manejadores de compensación:** Permite deshacer el trabajo que ya ha sido realizado cuando se produce un fallo en la ejecución de un proceso.
   - **Manejadores de terminación:** Permite especificar las actividades a llevar a cabo si se debe terminar la ejecución de un *scope*.
   - **Manejadores de eventos:** A través de los elementos *onEvent* determinan las acciones a realizar cuando se produce un evento dado, y mediante un elemento *onAlarm* las actividades a realizar si después de un tiempo determinado no se ha producido ningún evento.
4. Descripción del comportamiento del proceso de negocio a través de las actividades que proporciona el lenguaje.

Todos los elementos anteriores son globales por omisión. Aunque se tiene la posibilidad de declararlos de forma local utilizando el contenedor *scope*, con el que se puede dividir el proceso de negocio en diferentes ámbitos.

Los principales elementos que constituyen un proceso WS-BPEL son las actividades, que pueden tener atributos asociados y una serie de contenedores, que también pueden tener atributos asociados e incorporar distintos elementos. Las actividades pueden ser de dos tipos: básicas y estructuradas.

- **Actividades básicas:** Estas actividades tienen una labor específica en el proceso de negocio (recepción de mensajes, invocación de servicios, manipulación de datos, etc.).

- **Actividades estructuradas:** Estas actividades están compuestas por otras actividades y son las que definen la lógica del proceso de negocio.

Las distintas actividades que nos podemos encontrar en el lenguaje WS-BPEL 2.0 son las siguientes:

- **<receive>:** Mediante esta actividad se permite al proceso de negocio esperar para la llegada de un mensaje entrante.
- **<reply>:** Esta actividad permite que el proceso de negocio pueda enviar un mensaje en respuesta a uno entrante.
- **<invoke>:** Esta actividad permite al proceso de negocio invocar a una operación ofrecida por un socio.
- **<assign>:** Actividad utilizada para actualizar los valores de las variables.
- **<throw>:** Esta actividad se utiliza para generar un fallo desde el proceso de negocio.
- **<exit>:** Esta actividad se utiliza para salir inmediatamente del proceso de negocio.
- **<wait>:** Actividad utilizada para esperar un período de tiempo dado o hasta que se alcance un cierto punto en el tiempo.
- **<empty>:** Se trata de una actividad mediante la que no se realiza nada, pero puede ser útil para la sincronización de actividades concurrentes.
- **<sequence>:** Actividad que se utiliza para definir un conjunto de actividades que se realizarán de forma secuencial.
- **<if>:** Mediante esta actividad es posible seleccionar una determinada actividad a ejecutar entre un conjunto de opciones.
- **<while>:** Proporciona la ejecución repetida de la actividad hija que contiene mientras que la condición especificada sea verdadera.
- **<repeatUntil>:** Proporciona la ejecución repetida de la actividad hija que contiene hasta que la condición especificada sea verdadera.
- **<forEach>:** Proporciona la ejecución repetida de la actividad <*scope*> hija que contiene exactamente N+1 veces, donde N es el valor correspondiente a <*finalCounterValue*> menos el valor de <*startCounterValue*>. Esta actividad consta del atributo *parallel*, que si toma el valor "yes", se ejecutarían paralelamente las N+1 instancias del <*scope*> contenido.
- **<pick>:** Se trata de una actividad mediante la que, a través de los elementos <*onMessage*> permite determinar las actividades a realizar cuando se recibe un determinado mensaje y, a través de un elemento <*onAlarm*>, especificar las actividades a realizar cuando se alcance un determinado período de tiempo.
- **<flow>:** Esta actividad se utiliza para proporcionar concurrencia mediante la ejecución de todas sus actividades hijas en paralelo. También es posible definir dependencias de control sobre las actividades que contiene mediante los enlaces de sincronización.
- **<scope>:** Actividad que puede contener una actividad hija y permite la definición de forma local de los socios de negocio, variables y manejadores. Esta actividad tiene el atributo *isolated*, cuando dicho atributo toma el valor "yes" se protege el acceso a las variables compartidas, evitando que otro <*scope*> que utilice las mismas variables pueda acceder a ellas hasta que no haya terminado la ejecución del primer <*scope*>.

- **<compensate>:** Actividad utilizada para iniciar una compensación de todos los <*scope*> internos que ya han sido completados con éxito.

- **<compensateScope>:** Actividad utilizada para iniciar una compensación de un <*scope*> específico interno que ya ha sido completado con éxito.

- **<rethrow>:** Actividad utilizada para volver a lanzar un fallo que ha sido capturado por un manejador de fallos.

- **<validate>:** Esta actividad se utiliza para validar los valores de las variables con respecto a su definición de datos WSDL y XML asociada.

- **<extensionActivity>:** Esta actividad se utiliza para extender WS-BPEL mediante la introducción de un nuevo tipo de actividad.

A continuación, mostramos un ejemplo de código en lenguaje WS-BPEL 2.0:

```
<flow> ← Actividad estructurada
  <links> ← Contenedor
    <link name=
        "comprobarVuelo-A-reservarVuelo" ← Atributo/> ← Elemento
  </links>
  <invoke name="comprobarVuelo" . . . > ← Actividad básica
    <sources> ← Contenedor
      <source linkName=
          "comprobarVuelo-A-reservarVuelo" ← Atributo/> ← Elemento
    </sources>
  </invoke>
  <invoke name="comprobarHotel" . . . />
  <invoke name="comprobarAlquilerCoche" . . . />
  <invoke name="reservarVuelo" . . . >
    <targets> ← Contenedor
      <target linkName=
          "comprobarVuelo-A-reservarVuelo" /> ← Elemento
    </targets>
  </invoke>
</flow>
```

En el código anterior tenemos una actividad estructurada *flow* que contiene un conjunto de actividades *invoke* (*comprobarVuelo*, *comprobarHotel*, *comprobarAlquilerCoche* y *reservarVuelo*) que se ejecutarían en paralelo cuando la actividad *flow* comenzara. No obstante, en determinadas ocasiones puede que sea necesario establecer una sincronización entre algunas de estas actividades que están ejecutándose concurrentemente. En este caso, sólo tendría sentido reservar un vuelo una vez que se haya comprobado que dicho vuelo es correcto. Por lo tanto, es necesario establecer un enlace o *link* entre ambas actividades, introduciendo así, una dependencia entre ambas actividades que obligará a la actividad objetivo del enlace a ejecutarse solo cuando se haya completado la actividad fuente del enlace.

Por otro lado, en WS-BPEL pueden aplicarse distintos lenguajes de expresiones. Todos los motores WS-BPEL estándar soportan el lenguaje XML Path Language 1.1, se trata de un lenguaje declarativo con el que se puede realizar consultas sobre documentos XML y dispone de operadores aritméticos, relacionales y lógicos con una sintaxis similar a la de los lenguajes tradicionales.

1.3.4. BPELUnit

BPELUnit [5] es un marco de pruebas unitarias automáticas, repetibles y de caja blanca para probar composiciones WS-BPEL. Las pruebas se realizan mediante la especificación de unos determinados casos de prueba, que consisten en una secuencia de operaciones de entrada y salida que son ejecutadas por el marco BPELUnit en nombre del cliente y de todos los socios del proceso BPEL. Estos casos de prueba se definen en un fichero con extensión *bpts* (BPELUnit Test Specification).

Los ficheros con extensión *bpts* son ficheros XML con los que podemos especificar qué datos serán enviados y qué datos esperan ser recibidos por el cliente (*clientTrack*) y cada socio del proceso (*partnerTrack*), no siendo necesario que todos los socios interactúen en un determinado caso de prueba.

El elemento raíz de un fichero con extensión *bpts* se denomina *testsuite* y consta los siguientes elementos:

- **name:** Nombre con el se identifica al conjunto de pruebas.
- **baseURL:** BPELUnit utiliza una URL local para simular los socios del proceso. La URL de cada socio es la concatenación de la URL base más el nombre de cada socio. La URL base predeterminada es http://localhost:7777/ws.
- **deployment:** Contiene la información relativa al despliegue del proceso y consta de:
  - **put:** Detalla el proceso a desplegar mediante su nombre, el motor con el que se va a ejecutar y el fichero WSDL con la especificación del servicio.
  - **partner:** Mediante este elemento se especifican los socios del proceso, que pueden ser varios o ninguno. Cada socio se especifica mediante un nombre, con el que se identifica, y mediante el fichero WSDL que contenga su descripción.
- **testcases:** Elemento que contiene los distintos casos de prueba (*testcase*).

  Cada caso de prueba se identifica mediante un nombre. Además, un caso de prueba puede estar basado en otro definido anteriormente mediante el atributo *basedOn* y mediante el atributo *vary* se puede determinar si el caso de prueba se debe ejecutar varias veces. Por otro lado, mediante el atributo *abstract* se consigue que el caso de prueba no pueda ejecutarse al ser un caso de prueba abstracto.

  El contenido de los *clientTrack* y los *partnerTrack* consiste en una secuencia de actividades, que pueden ser las siguientes:

  - **sendOnly:** Mediante esta actividad se envía un único mensaje.
  - **receiveOnly:** Esta actividad espera un único mensaje y lo verifica.
  - **sendReceive:** Actividad con la que se envía un único mensaje, espera una respuesta síncrona y la verifica.
  - **receiveSend:** Esta actividad espera un único mensaje, lo verifica y envía una respuesta síncrona.
  - **sendReceiveAsync:** Mediante esta actividad se envía un único mensaje, se espera una respuesta asíncrona y se verifica.
  - **receiveSendAsync:** Actividad con la que se espera un único mensaje, lo verifica y envía una respuesta asíncrona.

Además, para la creación de los distintos casos de prueba, se puede utilizar el lenguaje de plantillas Apache Velocity, con el que se puede automatizar la generación de casos de pruebas.

En el siguiente fragmento de código podemos ver un ejemplo:

```
<?xml version="1.0" encoding="UTF-8"?>
<tes:testSuite xmlns:tri="http://TriangleProcess" xmlns:tes="http://www.
   bpelunit.org/schema/testSuite">
  <tes:name>suite-templates.bpts</tes:name>
  <tes:baseURL>http://localhost:7777/ws</tes:baseURL>
  <tes:deployment>
    <tes:put type="ode" name="Triangle">
      <tes:property name="ODEDeploymentServiceURL">http://localhost:8081/
         ode/processes/DeploymentService</tes:property>
      <tes:property name="DeploymentArchive">BpelUnit</tes:property>
      <tes:wsdl>TriangleArtifacts.wsdl</tes:wsdl>
    </tes:put>
  </tes:deployment>
  <tes:setUp>
    <tes:dataSource type="velocity" src="data.vm">
      <tes:property name="iteratedVars">elementA elementB elementC</
         tes:property>
    </tes:dataSource>
  </tes:setUp>
  <tes:testCases>
    <tes:testCase name="MainTemplate" basedOn="" abstract="false" vary="
       false">
      <tes:clientTrack>
        <tes:sendReceive service="tri:Triangle" port="TrianglePort"
           operation="process">
          <tes:send fault="false">
            <tes:template><![CDATA[<t:TriangleRequest xmlns:t="http://
               TriangleProcess">
  <t:a>$elementA</t:a>
  <t:b>$elementB</t:b>
  <t:c>$elementC</t:c>
</t:TriangleRequest>]]></tes:template>
          </tes:send>
          <tes:receive fault="false"/>
        </tes:sendReceive>
      </tes:clientTrack>
    </tes:testCase>
  </tes:testCases>
</tes:testSuite>
```

Tanto el sistema de generación de casos de pruebas basado en un algoritmo genético como la herramienta MuBPEL hacen uso del marco de pruebas unitarias BPELUnit, ya que es necesario para realizar la ejecución de las composiciones WS-BPEL.

1.3.5. Velocity

Velocity [6] consiste en un motor de plantillas basado en Java [7], que puede emplearse como una aplicación independiente con la que producir código fuente. Velocity puede ser utilizado para la creación de páginas web, SQL, PostScript y cualquier otro tipo de salida de plantillas. Fue creado para proporcionar una manera más simple, limpia y fácil de agregar contenido dinámico en una página web. La inclusión de dicho contenido dinámico se realiza mediante referencias, siendo una variable un tipo referencia.

Mediante la utilización de plantillas Velocity pueden generarse casos de pruebas automáticamente

que contengan las mismas actividades, pero siendo distinto el contenido de los mensajes en cada caso de prueba. El contenido de los distintos mensajes puede formarse a través de la utilización de ciertas variables, pudiendo obtenerse del mismo fichero, o de un fichero externo, los datos para las variables en cada caso de prueba.

En este proyecto también se ha empleado Velocity para realizar una plantilla con la que generar un informe con formato *html* que contenga información relativa a la ejecución realizada por el algoritmo genético, como estádisticas, resultados, operadores aplicados, etc.

A continuación, veamos un ejemplo de un enunciado del lenguaje de plantillas Velocity:

```
#set ($par = "false")
```

Todos los enunciados comienzan con el carácter #, que contiene una directiva, que se utilizan para realizar una determinada acción. En el caso del ejemplo se trata de la directiva *set*, que sirve para establecer el valor de una referencia, que comienzan con el carácter *$*.

1.3.6. Java

Java [7] es el principal lenguaje que se ha empleado para el desarrollo de este proyecto. Se trata de un lenguaje de programación orientado a objetos creado en 1995 por Sun Microsystems. Fue creado con la intención de que fuera independiente del sistema operativo, en este sentido el código Java se ejecuta en una máquina virtual.

Tiene una sintaxis parecida a C++, pero carece de los aspectos de bajo nivel que suelen provocar más fallos. Además, dispone de un recolector de basura que se encarga de borrar los objetos a los que no hay ninguna referencia, por lo que evita problemas de fuga de memoria y facilita el trabajo del programador.

1.3.7. JUnit

JUnit [8] es un *framework* que permite la creación de pruebas unitarias repetibles para el lenguaje de programación Java. Este conjunto de bibliotecas permite la prueba de módulos determinados de un programa de una manera sencilla y controlada. De esta manera, JUnit nos facilita la evaluación por separado del funcionamiento de todos los métodos de una clase mediante la aplicación de pruebas unitarias.

Una de las principales facilidades que ofrece JUnit es la proporción de asertos para tipos primitivos y estructuras que contengan estos tipos, de manera que permiten la comprobación de valores obtenidos con respecto a los esperados y opcionalmente emitir una cadena en la que se indique el fallo cometido. En el siguiente fragmento de código podemos ver un ejemplo:

```
@Test
public void testDoMutationIntFirst() {
        final BigInteger oldValue = new BigInteger("10011100010000",2);
                    //10000
        final BigInteger minValue = new BigInteger("-11000011010100000",2);
                   //-100000
        final BigInteger maxValue = new BigInteger("11000011010100000",2);
                    //100000
        final TypeInt type = new TypeInt(minValue, maxValue);
        final double probability = 2.0;
        final BigInteger newValue = mutation.doMutationInt(probability,
           type, oldValue);
        assertTrue("The mutation should have produced a new object with a
           different value", !oldValue.equals(newValue));
```

```
10          assertTrue("The mutated value should respect minimum value
               constraints", newValue.compareTo(minValue) >= 0);
11          assertTrue("The mutated value should respect maximum value
               constraints", newValue.compareTo(maxValue) <= 0);
12  }
```

Otro de los elementos útiles de JUnit es *@Before*, que permite la ejecución del código común necesario antes de los tests. Podemos observar un ejemplo en el siguiente código mostrado.

```
1   @Before
2   public void setUp() {
3           mutation = new GrayBinaryMutationOperator();
4           mutation.setPRNG(new Random(0));
5   }
```

1.3.8. HTML 5

HTML 5 es un lenguaje de marcado empleado para la creación de páginas web. Fue desarrollado por el World Wide Web Consortium. Este lenguaje determina una estructura principal y se escribe mediante el empleo de etiquetas.

En este proyecto, HTML 5 ha sido empleado junto a JavaScript para la generación de informes en los que se puede observar de una forma clara y rápida, los resultados obtenidos.

1.3.9. JavaScript

JavaScript es un lenguaje de programación interpretado. Fue creado por Brendan Eich. Se trata de un lenguaje utilizado normalmente en programación web para realizar operaciones del lado del cliente.

En este proyecto, ha sido empleado para la generación de informes junto a HTML 5, con la intención de presentar los resultados obtenidos por la herramienta de una manera más clara.

Capítulo 2

Planificación

En este capítulo se especifican cada una fases en las que se ha dividido el desarrollo de este proyecto fin de carrera, así como la metodología que se ha seguido para la realización del mismo.

El desarrollo de este proyecto ha tenido lugar desde julio de 2014 hasta septiembre de 2015.

2.1. Metodología

La metodología empleada durante el desarrollo de este proyecto fin de carrera ha sido la metodología iterativa basada en prototipos. Esta metodología facilita la interacción con los investigadores del grupo UCASE, ya que permite obtener un producto software utilizable al final de cada prototipo.

En este proyecto fin de carrera, se realizan los prototipos del sistema a medida que se amplia o corrige la funcionalidad del mismo y se añaden nuevos operadores genéticos, tanto de mutación como de cruce. Los resultados obtenidos a través de los distintos experimentos realizados también sirven como evaluación del sistema y para que los investigadores del grupo UCASE determinen las necesidades que consideran oportunas. De esta manera, a cada prototipo se le va añadiendo la funcionalidad necesaria de acuerdo al consenso alcanzado en los distintos puntos debatidos en el grupo de investigación UCASE.

2.2. Etapas del proyecto

2.2.1. Elicitación de requisitos

En esta etapa, se tuvieron las primeras reuniones con el grupo de investigación UCASE para que me plantearan las características de este proyecto fin de carrera.

En sucesivas reuniones, se fueron planteando los distintos requisitos que debía cumplir el proyecto, siguiendo el algoritmo génetico planteado por los investigadores del grupo UCASE. Así mismo, mediante el análisis de los resultados que se obtenían en los diversos experimentos desarrollados a lo largo del proyecto y las conclusiones obtenidas en los debates con el grupo de investigación, se determinaban nuevas funcionalidades que el sistema debía cumplir.

2.2.2. Estudio del dominio y de las tecnologías necesarias

En esta etapa del proyecto se realiza una fase de aprendizaje sobre el funcionamiento de los algoritmos genéticos, para tener una primera toma de contacto con la materia que se trata en este proyecto y así poder afrontar el desarrollo del mismo.

También se realiza una evaluación de todas las herramientas y tecnologías que iban a ser necesarias para todos los aspectos a desarrollar en este proyecto.

2.2.3. Fase de auditoría del algoritmo genético

Esta etapa del proyecto fin de carrera se trata de una fase de auditoría del código implementado, con el objetivo de detectar las desviaciones entre el algoritmo pretendido por los autores del grupo de investigación UCASE y lo que realmente estaba implementado en el sistema.

Esto permite además que se pueda conseguir una definición refundada del algoritmo y una implementación de acuerdo a dicha definición, de manera que el sistema ofrezca el funcionamiento deseado por los investigadores del grupo UCASE.

2.2.4. Desarrollo del algoritmo genético

En esta etapa se realiza el desarrollo del algoritmo genético, otorgándole el comportamiento que los investigadores del grupo UCASE desean, aumentando su funcionalidad y desarrollando operadores genéticos de cruce y mutación. De manera que el sistema posibilite la obtención de resultados científicos útiles para la realización de estudios de investigación.

2.2.5. Experimentos y pruebas

Esta etapa consta de la realización experimentos y pruebas para evaluar y mejorar el desarrollo llevado acabo en este proyecto.

También se han realizado experimentos con el objetivo de realizar estudios de investigación. En este sentido, se ha presentado un artículo a las XX Jornadas de Ingeniería del Software y Bases de Datos, llamado «Análisis y determinación del impacto del operador de mutación en la generación genética de casos de prueba para WS-BPEL», que ha sido aceptado para su publicación.

2.2.6. Visualización

La visualización de los resultados obtenidos en la ejecución del algoritmo genético se trata de algo de vital importancia, ya que es conveniente que los investigadores puedan analizar de manera sencilla los datos resultantes para poder extraer conclusiones sobre ellos sin tener que emplear demasiado tiempo procesando los resultados.

Es por ello que en esta etapa se mejora la visualización de los resultados obtenidos, estableciendo para ello la generación automatizada de informes en HTML 5 y JavaScript, en los que se puede observar de una manera clara y concisa las estadísticas y los resultados que ha producido el algoritmo durante una determinada ejecución.

2.2.7. Documentación

En esta etapa ha tenido lugar la realización de esta memoria, que se ha desarrollado haciendo uso del lenguaje LaTeX [9], ya que es de gran utilidad para desarrollar documentos de grandes dimensiones.

2.3. Distribución temporal

En la figura 2.1 podemos apreciar el Diagrama de Gantt, que muestra una representación gráfica de la planificación temporal para cada una de las distintas etapas de este proyecto fin de carrera.

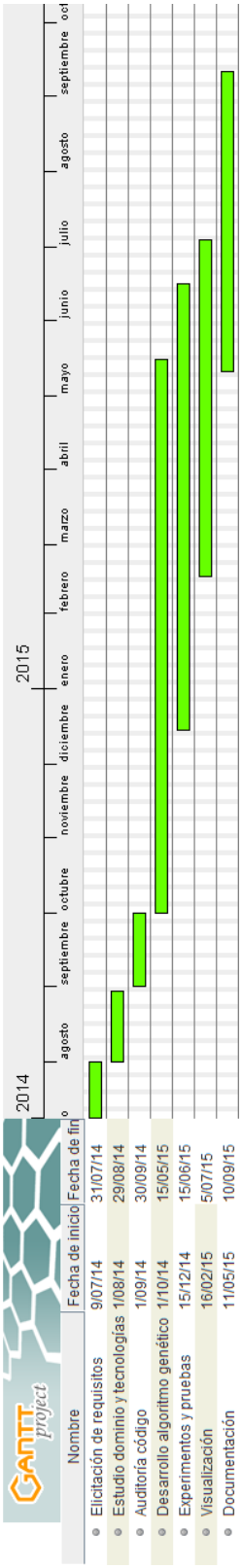


Figura 2.1: Planificación temporal.

Capítulo 3

Análisis

En este capítulo se detalla la fase de análisis del proyecto fin de carrera. Por lo que se especifican los requisitos funcionales y de implementación del sistema, así como la descripción del algoritmo a implementar deseado por los autores y el estado inicial que presentaba la herramienta.

3.1. Requisitos funcionales

Los requisitos funcionales de este proyecto son los siguientes:

- Implementar el algoritmo genético especificado por los autores. [10].
- Generar informes acerca del rendimiento del sistema. Como pueden ser:
  - Tiempo y espacio.
  - Calidad de los conjuntos de casos de prueba generados por el algoritmo genético.

3.2. Requisitos de implementación

El proyecto a desarrollar debe tener las siguientes características:

- Estará desarrollado con el lenguaje de programación Java.
- Será un proyecto de software libre.
- Estará bajo el sistema de control de versiones Subversion [11] en la forja empleada por el grupo UCASE.

3.3. Descripción del algoritmo deseado por los autores

Los autores del grupo de investigación UCASE proponen una técnica evolutiva basada en un algoritmo genético para la generación de casos de prueba para un sistema de mutaciones, con el objetivo de matar al mayor número posible de mutantes generados a partir de un determinado programa original a estudiar.

El generador de casos de prueba que forma parte de dicho sistema de mutaciones consta de un preprocesador y un algoritmo genético. El preprocesador encapsula todos los aspectos del código dependiente del lenguaje de programación del programa original a estudiar y produce toda la información necesitada por el algoritmo genético para generar los casos de prueba.

El algoritmo que debe ser implementado en este sistema viene descrito en la tesis que se cita [10]. Dicho algoritmo es totalmente independiente del lenguaje de programación en el que está escrito el programa original y produce generación a generación nuevos casos de prueba con el objetivo de matar al mayor número posible de mutantes obtenidos a partir del programa original. El procedimiento que se sigue para ello es el siguiente:

1. Se proporciona al algoritmo genético una especificación [12] con la estructura de los casos de prueba a generar y un conjunto de casos de prueba de partida que constituirá la población inicial del algoritmo genético.

2. Generación a generación el algoritmo genético va produciendo nuevos casos de prueba, de manera que los individuos que conforman la población de la siguiente generación son generados a partir de pares de individuos de la generación anterior mediante la aplicación de operadores genéticos de cruce y mutación.

   Los individuos de la generación anterior que se van a emplear para conformar la nueva generación se seleccionan a través de un operador de selección. Dicha selección se realiza teniendo en cuenta los valores de la función de aptitud para los casos de prueba generados.

   La función de aptitud empleada se basa tanto en el número de mutantes del programa original que mata un determinado caso de prueba como en el número de casos de prueba que matan a esos mutantes. De esta manera, se consigue distinguir casos de prueba especializados.

   Un individuo de una generación originado a partir de individuos de la generación anterior puede haber sufrido:

   - Cruce.
   - Mutación.
   - Cruce y mutación.
   - Ninguna de ellas, por lo que el individuo sería idéntico a su ascendiente.

   Podemos observar el comportamiento que debe seguir el algoritmo genético para generar un par de individuos a partir de la generación anterior en el algoritmo 1. Dicho comportamiento se puede resumir de la siguiente manera:

   - Por cada par de individuos seleccionados de la población anterior se aplica cruce si al generar un número aleatorio n es menor que la probabilidad de cruce p_c.
   - A cada individuo se le aplica mutación si al generar un número aleatorio n es menor que la probabilidad de mutación p_m.
   - Este proceso se repite hasta que la población actual tenga un tamaño idéntico a la anterior.

   Se realizan las generaciones necesarias hasta que se cumpla una determinada condición de parada.

3. La salida del generador de casos de prueba es el conjunto formado por todos los casos de prueba generados en cada una de las diferentes generaciones que ha realizado el algoritmo genético.

3.4. Estado inicial de la herramienta

Una de las primeras fases de este proyecto fin de carrera se trata de una fase de auditoría del código para poder detectar las desviaciones entre lo que los autores deseaban y lo que realmente estaba

Algoritmo 1 Generación de descencientes

```
1: {Selección con el método de la ruleta.}
2: padres ← selec-padres(población)
3: {Cruce con probabilidad p_c.}
4: si aleatorio-uniforme(0,1) < p_c entonces
5:    descendiente ← hacer-cruce(padres)
6: si no
7:    descendiente ← padres
8: fin si
9: {Mutación con probabilidad p_m.}
10: si aleatorio-uniforme(0,1) < p_m entonces
11:    descendiente ← hacer-mutación(descendiente)
12: fin si
13: devolver descenciente
```

implementado en el sistema. Además, de este modo se puede obtener posteriormente una definición refundada del algoritmo y una implementación de acuerdo a esa definición, de tal manera que se consiga una herramienta que ofrezca el funcionamiento deseado por los investigadores.

Se observaron discrepancias entre los resultados obtenidos y los esperados, por lo que se decidió revisar la implementación del herramienta [3]. Durante la fase de auditoría del código se detectaron bastantes diferencias entre la idea que tenían los autores sobre el comportamiento de la herramienta y lo que realmente estaba implementado.

Una de las principales diferencias que podemos encontrar en la implementación inicial de sistema y lo que los autores realmente querían, radica en el comportamiento que seguía el algoritmo genético a la hora de realizar cruces y mutaciones. En este sentido, en el algoritmo genético inicial los cruces y mutaciones se realizaban de un modo exclusivo, es decir, solo se podía aplicar un operador genético a un determinado individuo, algo que los autores no deseaban, puesto que no se ajustaba a lo que necesitaban para obtener los resultados de investigación necesarios para poder realizar trabajos y artículos de investigación. La idea de los investigadores era que se pudiera aplicar a cada individuo cruce y mutación, solo cruce, solo mutación o ninguno de ellos.

De acuerdo a lo anterior, en la implementación inicial del sistema no se trataba de manera independiente la decisión de cruzar y mutar. Por lo que no se ajusta al algoritmo deseado por los autores, ya que solamente se realizaba la generación de un número aleatorio para comparar con la probabilidad de aplicación de cada uno de los operadores genéticos y decidir cual de ellos aplicar. En este sentido, además los operadores de cruce y mutación se trataban como un mismo operador genético, lo que dificulta la diferenciación entre operador de mutación y operador de cruce a un nivel abstracto y además, complica la distinta aplicación que requieren unos y otros en el algoritmo genético.

Por otro lado, en la implementación inicial del sistema no se garantizaba que la selección se realizase de un número par de individuos. Lo que podía llevar a errores, ya que para realizar la reproducción de individuos, el algoritmo genético debe utilizar en cada generación pares de individuos de la generación anterior a los que se les puede aplicar los operadores genético de cruce y mutación para pasar a conformar la población de la nueva generación. Además, de este modo, el algoritmo no podría garantizar que el número de individuos que constituyen una población se mantenga idéntico durante las distintas generaciones que realiza el algoritmo genético, puesto que se podría seleccionar un número de individuos menor al requerido.

Además, la herramienta solo disponía de un operador de mutación y otro de cruce, por lo que presentaba una pronunciada escasez de variedad de operadores de cruce y mutación. Lo que hacía que los investigadores no tuvieran mucha libertad de movimientos a la hora de plantear sus estudios, ya que es

necesario contar con un amplio abanico de operadores entre los que elegir, tanto de cruce como de mutación, para que así los investigadores puedan hacer hincapié en la comparación entre ellos y, además, les permite poder elegir el que crean más conveniente para cada problema en concreto.

Así mismo, el operador de mutación implementado mutaba una componente elegida al azar entre las n posibles que componen el individuo. Por lo que la probabilidad de que se mute una componente concreta en una generación es p_m/n. Este método presenta un problema potencial que puede hacer que la mutación no sea todo lo eficaz que debiera, si entendemos que el objetivo de la mutación es introducir una diversidad en las poblaciones que no puede conseguirse mediante cruce, ya que de esta manera habrá componentes que tendrán muy pocas posibilidades de sufrir cambios.

Uno de los inconvenientes a tener muy en cuenta también del estado inicial de la aplicación, es que los pasos y resultados que se podían dar durante una ejecución del algoritmo genético, se mostraban de una manera muy básica y de difícil comprensión a primera vista. Este hecho provocaba que para los investigadores resultara una tarea ardua saber el comportamiento que ha seguido el algoritmo genético durante una ejecución, así como el hecho de analizar sus resultados, puesto que no se disponía de una visualización que permitiera observar de una manera rápida y sencilla estos aspectos.

3.5. Tareas a realizar

Con la reingeniería de esta herramienta se pretende solventar las discrepancias existentes entre el algoritmo definido por los autores y el estado actual del sistema. Además, durante el estudio de la herramienta se han detectado algunas posibles mejoras que se han discutido con los integrantes del grupo de investigación para realizar su desarrollo.

3.5.1. Tareas relacionadas con la revisión de la definición del algoritmo

- Modificar el comportamiento del algoritmo genético con respecto a la reproducción de los individuos en cada generación. Es decir, en lugar de aplicar cruce y mutación de una manera mutuamente excluyente, establecer que en cada generación a los individuos seleccionados se les pueda aplicar:
  - Cruce.
  - Mutación.
  - Cruce y mutación.
  - Ninguna de ellas.
- Además, también es necesario tratar la decisión de aplicación de los cruces y mutaciones de manera independiente, empleando la generación de números aleatorios por separado para decidir la aplicación de los operadores de cruce y mutación de acuerdo a la probabilidad de aplicación de cada uno.
- En este sentido, se debe introducir en el comportamiento del algoritmo genético la responsabilidad de decidir qué componentes de un determinado individuo deben ser mutadas, en lugar de decidir únicamente si el individuo en sí debe ser mutado. De este modo, se exime a los operadores de mutación de decidir que componentes del individuo en cuestión deben ser mutadas.
- Por otro lado, también es necesario garantizar que en la selección de los individuos que se van a utilizar para conformar la población de la siguiente generación se realiza la selección de un número par de individuos, ya que la reproducción en cada una de las generaciones del algoritmo genético se realiza a través de parejas de individuos de la generación anterior. Y así, poder garantizar

además que el número de individuos que conforman una población se mantiene idéntico de una generación a otra.

- Debido a la escasez de operadores de cruce y mutación, es necesario dotar a la herramienta con un abanico más amplio de ambos operadores.

3.5.2. Tareas relacionadas con la revisión de la implementación

- Evitar que los operadores de cruce y mutación se traten como un mismo operador genético, es decir, que no se implementen bajo la misma interfaz. De esta manera, es necesario que los operadores de cruce implementen una determinada interfaz y los de mutación otra.
- Controlar en el comportamiento del algoritmo genético que no se modifican los individuos de una generación mientras se construye la siguiente.
- Permitir la elección del operador de mutación para cada tipo, así como para cada posición, mediante el empleo del patrón de diseño *Composite* [13].
- Realizar algunas modificaciones en la implementación de los operadores genéticos existentes para garantizar el correcto funcionamiento de los mismos.

3.5.3. Tareas relacionadas con la revisión de la generación de informes

- Realizar una visualización con HTML 5 [14] y JavaScript en la que se muestre de una manera detallada los principales aspectos a analizar de los resultados de una ejecución del algoritmo genético. De este modo, los investigadores pueden sacar conclusiones de una manera más fácil y rápida, observando la información en el fichero con extensión *html* sin necesidad de tratar los resultados previamente.
- Mejora del *logger* de texto aumentando la información a generar y mostrándola de una manera más detallada.

Capítulo 4

Diseño y reingeniería

En este capítulo se trata el diseño del proyecto y la reingeniería realizada. En este sentido se detallan cada uno de los aspectos en los que se ha centrado la reingeniería del sistema. Además, se realiza una definición refundada del algoritmo y se especifica la estructura que se sigue para la generación de casos de pruebas.

4.1. Reingeniería

4.1.1. Revisión de la definición del algoritmo genético

4.1.1.1. Aplicación operadores de cruce y mutación

Con la reingeniería de esta herramienta se ha conseguido solventar el problema que tenía originalmente el comportamiento seguido por el algoritmo genético, ya que aplicaba los operadores genéticos de un modo exclusivo a un determinado individuo, es decir, a efectos prácticos no se podía aplicar a la vez cruce y mutación a un determinado individuo, sino que se aplicaba solo cruce, o solo mutación o ninguna de las dos cosas.

En este sentido, se ha implementado el algoritmo genético de manera que permita aplicar a cada individuo:

- Cruce.
- Mutación.
- Cruce y mutación.
- Ninguna de las dos.

A efectos prácticos, con la nueva implementación realizada, podemos diferenciar dos comportamientos del algoritmo genético, según se especifique el uso de un operador o varios de cada tipo, cruce y mutación.

Teniendo en cuenta que p_c es la probabilidad de aplicación de un determinado operador de cruce.

- Si solo se ha especificado un operador de cruce el procedimiento seguido es el siguiente:
  - Se genera un número aleatorio y uniformemente distribuido en el intervalo [0, 1].
  - Si $y < p_c$, se aplica el operador de cruce determinado para producir los nuevos individuos.
  - En caso contrario no se aplica cruce y los nuevos individuos generados son idénticos a sus padres.

- Si se han especificado varios operadores de cruce el procedimiento seguido es el siguiente:
  - Se genera un número aleatorio y uniformemente distribuido en el intervalo [0, 1].
  - Si $y < p_c$, se aplica el operador de cruce determinado para producir los nuevos individuos.
  - En caso contrario a y se le resta la probabilidad de cruce del operador actual, $y = y - p_c$, y se compara con la probabilidad de cruce del siguiente operador para determinar si se aplica dicho operador. Esto es así ya que las probabilidades de los distintos operadores de cruce se toman como intervalos en los que se puede encontrar y: [0, p_{c1}), [p_{c1}, $p_{c1} + p_{c2}$), ..., [$\sum_{i=1}^{n-1} p_{ci}$, $\sum_{i=1}^{n} p_{ci}$).
    Este procedimiento se repite hasta que se aplique un operador de cruce o hasta que no haya más operadores de cruce. En este último caso, los nuevos individuos serán idénticos a sus padres.

En realidad, tal y como se puede observar, el procedimiento seguido cuando se ha especificado un solo operador de cruce no es más que un caso particular del procedimiento seguido cuando se han especificado varios operadores.

Teniendo en cuenta que p_m es la probabilidad de aplicación de un determinado operador de mutación.

- Si solo se ha especificado un operador de mutación el procedimiento seguido es el siguiente:
  - Para cada hijo se recorre con un bucle cada una de sus componentes.
  - Para cada componente se genera un número aleatorio z uniformemente distribuido en el intervalo [0, 1].
  - Si $z < 1 - \sqrt[n]{1 - p_m}$, se muta la componente de acuerdo a su tipo.
- Si se han especificado varios operadores de mutación el procedimiento seguido es el mismo. No obstante, si en un individuo no se ha mutado ninguna de sus componentes se repetirá el mismo proceso con dicho individuo empleando el siguiente operador de mutación.

  Este procedimiento se repite hasta que en dicho individuo se aplique un operador de mutación o hasta que no haya más operadores de mutación.

Del mismo modo, el procedimiento seguido cuando se ha especificado un solo operador de mutación no es más que un caso particular del procedimiento seguido cuando se han especificado varios operadores.

La fórmula empleada para la decisión de mutar una componente es indispensable para garantizar que la probabilidad de mutación de un individuo sea exactamente p_m y que todas sus componentes tengan la misma probabilidad de ser mutadas.

$$z \in [0, 1], \quad z < 1 - \sqrt[n]{1 - p_m} \tag{4.1}$$

.

Podemos ver un diagrama del comportamiento general del algoritmo genético en la figura 4.1.

4.1.1.2. Selección de un número par de individuos

Es necesario garantizar que la selección de individuos que van a emplear en el proceso de reproducción se realiza sobre un número par. Ya que para realizar la reproducción de individuos correctamente, generación a generación el algoritmo genético produce los individuos de la población de la siguiente generación mediante el empleo de pares de individuos de la generación anterior que pueden sufrir la

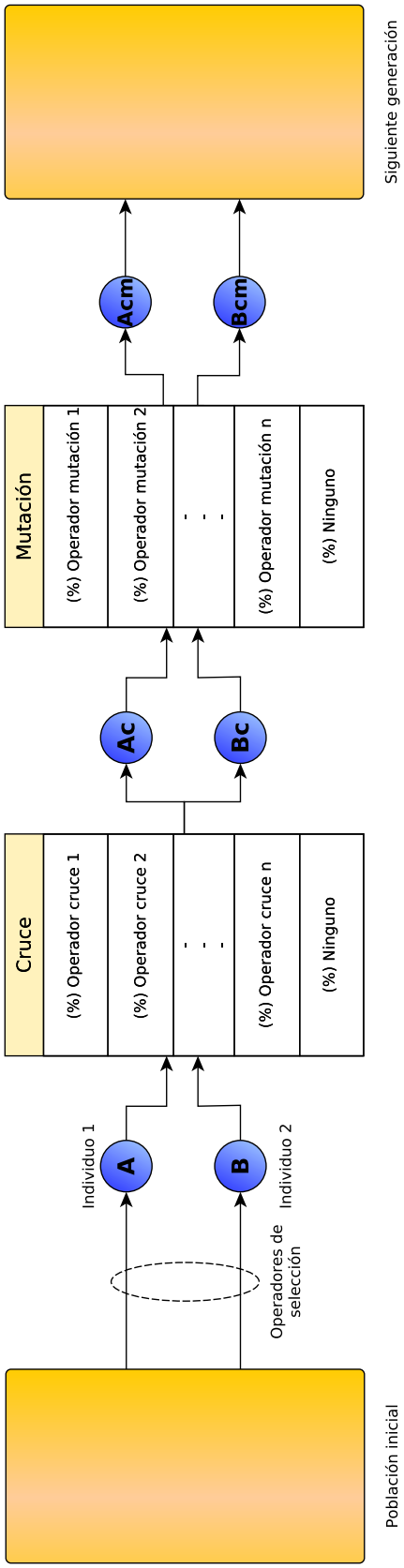


Figura 4.1: Comportamiento general del algoritmo genético.

aplicación de los operadores de cruce y mutación. De esta manera también se puede garantizar que el número de individuos que conforman la población se mantiene idéntico de una generación a otra.

La selección de individuos se realiza de la siguiente manera:

- Si el número n de individuos a seleccionar es impar, se selecciona $n + 1$ elementos.
- Se recorre cada uno de los operadores de selección especificado. Cada uno de ellos selecciona un determinado número de individuos según las siguientes condiciones:
  - Si se trata del último operador de selección, se seleccionan el número de individuos que faltan para completar el número total de individuos que deben ser seleccionados.
  - En caso contrario, se selecciona el mínimo entre la cantidad de individuos que faltan y la cantidad de individuos que corresponde al porcentaje sobre la cantidad total de individuos redondeado del operador.

4.1.1.3. Aumento del número de operadores de cruce y mutación

En esta nueva implementación de la herramienta, se han añadido una amplia variedad de operadores de cruce y mutación. Lo que facilita el trabajo de los investigadores ya que de esta manera pueden elegir el operador que consideren más apropiado para un determinado problema en concreto, e incluso pueden compararlos entre ellos para estudiar cual obtiene mejores resultados.

Los operadores de cruce que se han añadido son:

- Operador de cruce de doble punto.
- Operador de cruce uniforme.

Los operadores de mutación que se han añadido son:

- Operador de mutación *1-Bit Flip*.
- Operador de mutación en código de *Gray*.
- Operador de mutación específico para la mutación de triángulos.
- Operador de mutación de listas con probabilidad personalizada.
- Operador de mutación *Random Resetting*.
- Operador de mutación *Composite* para elegir mutación por tipos.
- Operador de mutación *Composite* para elegir mutación por posiciones.

Puede ver la descripción de todos los operadores genéticos disponibles en la sección 5.

4.1.2. Revisión de la implementación

4.1.2.1. Separación de los operadores de cruce y mutación

En la implementación inicial los operadores de cruce y mutación se implementaban bajo la misma interfaz, ya que ambos se trataban como operadores genéticos sin realizar ningún tipo de distinción entre ambos. En la implementación actual se han separado las interfaces de los operadores de cruce y mutación, ya que son operadores que se comportan de distinta manera. Además, el operador de cruce

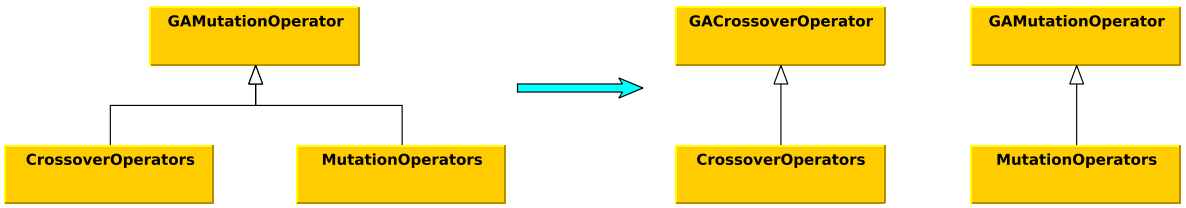


Figura 4.2: Comparación de las interfaces de los operadores de cruce y mutación antes y después.

opera sobre dos individuos mientras que el operador de mutación opera sobre un solo individuo, en lugar de que ambos reciban y devuelvan poblaciones como ocurría en la implementación inicial.

De esta manera, se facilita la aplicación por separado de ambos operadores en el algoritmo genético y mejora la distinción de ambos por parte de los investigadores a la hora de utilizar la herramienta y realizar estudios.

En la figura 4.2 podemos ver la diferencia en la estructura de los operadores de cruce y mutación antes y después.

4.1.2.2. Mantener inalterables los individuos de una población

Hay que prestar especial atención a no modificar los individuos de una generación mientras se construye la siguiente. Por ello se construyen copias de los individuos a medida que el algoritmo genético va tomando parejas de individuos para conformar la población de la nueva generación. De esta manera, se garantiza que solamente se modifican a causa de la aplicación de los operadores de cruce y mutación los individuos de la siguiente generación, dejando intactos los individuos que conforman la población de la generación previa.

4.1.2.3. Mutación según tipos de datos y posiciones

Poder usar solamente un operador de mutación para todos los tipos de datos que comprenden un problema a estudiar, limita mucho el margen de maniobras a la hora de poder realizar el estudio de investigación. Es por ello que se ha implementado dos operadores de mutación que emplean el patrón de diseño *Composite*. Mediante el empleo dicho patrón podemos especificar varios operadores de mutación siguiendo un determinado criterio, gracias a su estructura en forma de árbol y a la composición recursiva.

Con esos nuevos operadores de mutación podemos:

- Especificar que operador se quiere emplear para cada tipo de datos.
- Especificar que operador se quiere emplear para cada posición.

Además, también podemos indicar que operadores de mutación queremos emplear para los distintos elementos contenidos en tuplas y en listas.

4.1.2.4. Mejoras de operadores genéticos ya existentes

Para garantizar el correcto funcionamiento de los operadores genéticos ha habido que realizarles algunos cambios. Como que en vez de trabajar con poblaciones de individuos, se apliquen directamente sobre los individuos implicados, así como en sus componentes.

Además, en los operadores de cruce se devuelven las posiciones de las componentes implicadas en el intercambio. Por otro lado, en la mutación se realiza copia de seguridad de las listas para facilitar su seguimiento en los informes generados.

4.1.3. Revisión de la generación de informes

4.1.3.1. Generación de informe html

Los resultados que ofrecía el sistema se presentaban de una manera muy simple y poco sintetizada, lo que forzaba a que los investigadores tuvieran que realizar una ardua y lenta tarea al analizar los resultados obtenidos por el algoritmo genético para llegar a alguna conclusión.

En este sentido, se ha realizado una visualización haciendo uso HTML 5 y Javascript en el que se puede observar los principales resultados y estadísticas de una ejecución del algoritmo genético de una manera detallada y de fácil comprensión sin necesidad de tratar los datos obtenidos previamente.

Los aspectos que se muestran en en el informe con extensión html son:

- Estadísticas generales de la ejecución:
  - Acumulación de mutantes distintos del programa original que han sido matados a lo largo de las generaciones de la ejecución.
  - Resultados de los mutantes del programa original en cada generación.
  - Aptitud de los individuos de cada generación.
  - Condición de terminación de la ejecución.
- Resultados específicos de cada generación:
  - Nuevos mutantes generados.
  - Cruces aplicados.
  - Mutaciones aplicadas.
  - Población de la generación.
  - Matriz de ejecución.

4.1.3.2. Ampliación de informe de texto

Además, para poder albergar más información en el *logger* de texto ya existente, se ha aumentado la información que recoge. En este sentido se ha añadido información referente a los individuos generados, el inicio y la finalización de la ejecución de los individuos, los resultados de la matriz de ejecución, los individuos seleccionados, los cruces aplicados, las mutaciones aplicadas y la condición de terminación cumplida.

Además, se ha añadido una opción *verbose* con la que se puede especificar si se quiere un grado de detalle mayor o no.

4.2. Definición refundada del algoritmo

El algoritmo genético recibe una especificación con la estructura de los casos de prueba a generar, indicado en un fichero de configuración con formato YAML, donde se especifica además el resto de información necesaria que el sistema necesita como entrada para su ejecución, como el tamaño que debe tener la población, los distintos operadores a emplear, las probabilidades de cruce y mutación, los informes a generar y las condiciones de parada del algoritmo. También puede recibir un conjunto de casos de prueba de partida que constituirá la población inicial del algoritmo genético. El algoritmo genético producirá generación a generación nuevos casos de prueba mediante cruce y mutación de individuos, con el objetivo de obtener un conjunto de casos de prueba que mate al máximo número de mutantes posibles.

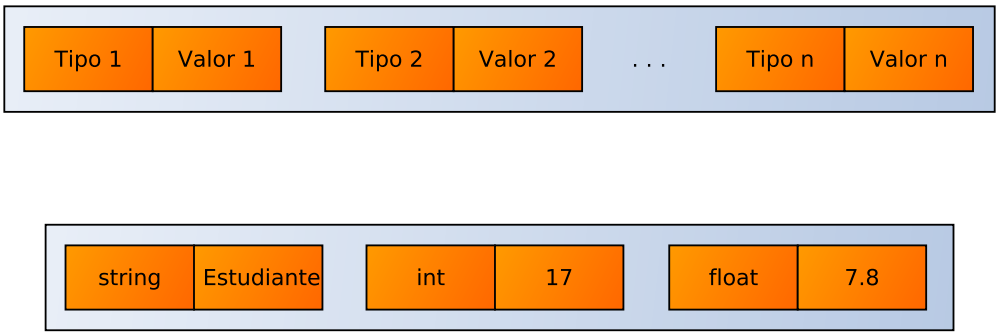


Figura 4.3: Codificación de los individuos.

Cuando el algoritmo genético acaba su ejecución al haber alcanzado algunas de las condiciones de terminación especificadas, se obtiene como resultado todos los casos de prueba que el algoritmo genético ha ido generando durante su ejecución, así como el resto de informes generados.

A continuación, se describe el procedimiento que sigue el algoritmo genético para generar un conjunto de casos de prueba que mate al mayor número posible de mutantes del programa original. Se especifica la codificación que siguen los individuos, la forma de obtener la población inicial que empleará el algoritmo genético, así como el procedimiento para calcular la aptitud de cada individuo y para producir nuevos casos de prueba generación a generación a través de cruces y mutaciones.

4.2.1. Codificación de los individuos

Cada uno de los individuos que conforma la población codifica un determinado caso de prueba. La estructura que siguen los individuos debe ser los más flexible posible para que pueda adaptarse a cualquier programa, debido a que los casos de prueba son dependientes en gran medida del programa que se desea estudiar.

En la figura 4.3 podemos observar la estructura empleada en la codificación de los individuos. Cada individuo está conformado por un vector de pares *clave-valor*, siendo la clave el tipo de la variable de una determinada componente y el valor un literal particular del tipo pertinente.

4.2.2. Primera población

En el algoritmo genético la población correspondiente a la primera generación se trata de una manera especial, puesto que no hay una población de una generación anterior en la que basarse para generar los individuos de la nueva población a través de cruces y mutaciones. Es por ello que hay que generar una primera población inicial con la que el algoritmo genético pueda producir las siguientes generaciones. Los individuos de esta población inicial se producen a través de generadores, capaces de crear estos nuevos individuos.

En este sentido, al sistema se le puede proporcionar un fichero en formato *vm* que contenga una población de partida. Si se proporciona dicha población se utilizará el generador *FixedGenerator* para producir cada uno de los individuos de la población inicial a partir de dicho fichero.

Por el contrario, si no se proporciona ninguna población de partida los individuos se crearán a partir de los generadores especificados.

Cada generador especificado tiene un porcentaje asociado que representa la proporción de individuos que debe producir con respecto al total de la población, generándose en cada caso el mínimo entre:

- el número redondeado de individuos que supone el porcentaje establecido para el generador.
- el número de individuos restantes para alcanzar el tamaño que debe tener la población.

Además, se puede especificar que un generador solo sea empleado para la generación de individuos de la primera población, por lo que no sería empleado en generaciones posteriores.

Los generadores disponibles son: *FixedGenerator* y *RandomGenerator*.

4.2.2.1. El generador *FixedGenerator*

El generador *FixedGenerator* crea individuos a partir una población dada en un fichero *vm*. El contenido del fichero es una secuencia de valores para cada una de las componentes que constituyen un individuo, el número de valores para cada componente coincide con el tamaño que debe tener la población.

Para generar cada individuo se recorre el fichero tomando un valor de cada una de las componentes que conformarán el nuevo individuo.

En la figura 4.1 podemos ver un ejemplo de un fichero *vm*.

Listado 4.1: Fichero con los valores de las componentes de los casos de prueba

```
1 #set($elementA = [400, 496, 1729, 1862, 1192])
2 #set($elementB = [715, 386, 1289, 1143, 554])
3 #set($elementC = [1558, 1235, 490, 1832, 1488])
```

4.2.2.2. El generador *RandomGenerator*

Se trata de un generador que produce nuevos individuos aleatoriamente. Los valores de cada componente que constituye al individuo se generan de acuerdo a una determinada estrategia (*IStrategy*) [15] especificado.

En este sentido, el caso más común es emplear un fichero *spec* que contenga los tipos de cada una de las componentes de las que puede estar formado un individuo. De esta forma, mediante la estrategia especificada se pueden generar los valores de cada componente de acuerdo a su tipo a partir de dicho fichero *spec*.

4.2.3. Cálculo de la aptitud

Una vez que se han generado todos los individuos de una población es necesario calcular la aptitud de cada uno de ellos. Para calcular la aptitud de cada individuo es necesario obtener la matriz de ejecución, que contiene la comparación de la ejecución del programa original frente a la ejecución de cada mutante contra cada uno de los casos de prueba que forman parte de la población.

En este sentido, para calcular dicha matriz el algoritmo divide la población entre individuos que ya se han ejecutado previamente e individuos que no se han ejecutado todavía, debido a que los resultados de los primeros ya se tienen y, por tanto, solo es necesario ejecutar los últimos. Cuando ya se han obtenido los resultados de aquellos individuos que no habían sido ejecutados todavía, se unen con los resultados de los individuos que ya lo habían sido para completar la matriz y poder obtener la aptitud de cada uno de ellos.

En el cálculo de la aptitud de cada individuo se tiene en cuenta el número de mutantes matados por el caso de prueba al que representa y el número de casos de prueba que matan al mismo mutante. De esta manera, se evita asignar una aptitud bajo a casos de prueba especializados. Por el contrario, si solo se tuviera en cuenta el número mutantes matados por el caso de prueba, se le podría asignar una aptitud

bajo a un caso de prueba que matase a un mutante que no puede ser matado por ningún otro caso de prueba, lo que no sería apropiado.

La fórmula para el cálculo de la aptitud es la siguiente. Siendo *M* el conjunto de mutantes generados del programa original, *T* el conjunto de casos de prueba que constituyen una determinada población y e_{ij} la diferencia en la ejecución del programa original con respecto al mutante $m_i \in M$ sobre el caso de prueba $t_j \in T$.

$$f(t_j) = \sum_{i=1}^{|M|} \frac{e_{ij}}{\sum_{k=1}^{|T|} e_{ik}} \tag{4.2}$$

Veamos un ejemplo de como calcular la aptitud de cada individuo a partir de una determinada matriz de ejecución $E = [e_{ij}]$, siendo $e_{ij} = 0$ cuando t_j no mata a m_i y $e_{ij} = 1$ cuando t_j mata a m_i.

$$E = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

.

Cada columna de la matriz representa un caso de prueba y sus celdas indican qué mutantes produjeron con él un resultado distinto respecto al programa original. Para calcular la aptitud de cada caso de prueba se procede de la siguiente manera:

- En primer lugar para cada fila se suma el valor de cada una de las celdas de dicha fila.
- A continuación para cada caso de prueba se recorre cada celda de la columna a la que representa y para cada una de las celdas se calcula una división en la que:
  - el numerador será el valor de la celda actual.
  - el denominador será la suma de todos los valores de la fila correspondiente a la celda actual.
- Por último se suma el resultado obtenido para cada celda de la columna y el resultado de dicha suma será la aptitud del caso de prueba correspondiente.

De esta manera, para la matriz dada obtendríamos los siguientes aptitudes para cada caso de prueba:

$$f(t_1) = 0/2 + 0/3 + 1/1 + 0/2 = 1$$

$$f(t_2) = 1/2 + 1/3 + 0/1 + 0/2 = 5/6$$

$$f(t_3) = 0/2 + 1/3 + 0/1 + 1/2 = 5/6$$

$$f(t_4) = 1/2 + 1/3 + 0/1 + 1/2 = 4/3$$

Como podemos observar, t_4 es el individuo con mayor aptitud puesto que es el caso de prueba que más mutantes mata. El siguiente individuo con aptitud mas alta es t_1, que solo mata a un mutante, ya que es un caso de prueba más especializado al ser el único que mata al mutante m_3. Por otro lado, t_2 y t_3 tienen la misma aptitud puesto que matan al mismo número de mutantes.

Una vez se ha calculado la aptitud se añade la generación al histórico y el algoritmo genético continua generando nuevos individuos generación a generación.

4.2.4. Generaciones

El algoritmo genético tiene como objetivo producir generación a generación nuevos casos de prueba con los que matar el mayor número posible de mutantes del programa original.

Para ello, una vez que se ha generado la población inicial, para cada generación posterior se genera en primer lugar un porcentaje de nuevos individuos a través de un generador, si se ha especificado alguno. Si se genera algún individuo de esta manera se añade directamente en la nueva población.

Cabe destacar que el tamaño de la población debe mantenerse constante a lo largo de todas las generaciones que realiza el algoritmo genético, por lo que el resto de individuos que formarán parte de la población de la próxima generación son generados a partir de pares de individuos de la generación anterior a los que se les puede aplicar cruce y mutación. En este sentido, mediante los operadores de selección que se hayan especificados se seleccionan de la población de la generación anterior los individuos necesarios para completar la nueva población, y mediante la aplicación de operadores de cruce y mutación se obtienen los nuevos individuos que conformarán la nueva población.

Una vez se han generado todos los individuos de una determinada generación se les calcula la aptitud §4.2.3 a cada uno de ellos y se añade la generación al histórico.

4.2.5. Selección

Para completar la población de la siguiente generación se seleccionan de la generación anterior los individuos necesarios a partir de operadores de selección. Cada operador de selección especificado lleva asociado un porcentaje que determina el número de individuos que dicho operador debe seleccionar de la población anterior. Es necesario controlar que no se seleccionen más individuos de los debidos, para ello cada operador seleccionará el mínimo entre:

- el número redondeado de individuos que supone el porcentaje establecido para el operador de selección.
- el número de individuos restantes para alcanzar el tamaño que debe tener la población seleccionada.

  En caso de que se trate del último operador de selección, siempre se seleccionará el número de individuos restantes para alcanzar el tamaño requerido.

Los individuos de la población anterior se seleccionan aplicando los operadores de selección especificados hasta alcanzar el tamaño que debe tener la población de individuos seleccionados. El número total de individuos seleccionados debe ser par, por lo que se seleccionará un individuo más en caso de que el número de individuos a seleccionar sea impar. Esto es así ya que los individuos seleccionados se van tomando de dos en dos, y a cada pareja de individuos se les puede aplicar cruce y mutación, solo cruce, solo mutación o ninguna de ellas antes de ser añadidos a la población de la siguiente generación. De esta manera, se consigue el objetivo de garantizar que el tamaño de la población se mantiene idéntico generación a generación. Posteriormente, los individuos resultantes se añaden a la nueva generación.

Los operadores de selección disponibles son: *RouletteSelection* y *UniformRandomSelection*.

4.2.5.1. El operador de selección *RouletteSelection*

Con el método de la ruleta se selecciona un individuo de la siguiente manera:

- Se calcula la suma t de las aptitudes de todos los individuos.
- Se genera un número aleatorio x uniformemente distribuido en el intervalo [0, t).

- Se genera un entero aleatorio p uniformemente distribuido en el intervalo [0, n), siendo n el número de individuos.
- Empezando por la posición p, se recorren los individuos acumulando su aptitud mientras que la suma parcial obtenida s sea menor que x.
- Se selecciona el individuo para el que s ha dejado de ser menor que x.

En este método, la probabilidad de selección de un individuo en cada operación de selección p_s es proporcional a su aptitud a. En concreto $p_s = a/t$, es decir, su aptitud normalizada al intervalo [0, 1].

Además, cada individuo se selecciona independientemente de los demás, por lo que se puede seleccionar varias veces al mismo individuo durante la creación de la población de la nueva generación.

4.2.5.2. El operador de selección *UniformRandomSelection*

Con este método se selecciona un individuo de la siguiente manera:

- Se genera un entero aleatorio p uniformemente distribuido en el intervalo [0, n), siendo n el número de individuos.
- Se selecciona el individuo que se encuentre en la posición p.

En este método la probabilidad de selección de un individuo en cada operación de selección p_s es inversamente proporcional al número de individuos que conforman la población. En concreto, $p_s = 1/n$.

Al igual que en el método de la ruleta cada individuo se selecciona independientemente de los demás, por lo que un mismo individuo puede ser seleccionado varias veces durante la creación de la población de la nueva generación.

4.2.6. Cruce

El cruce consiste en la generación de dos nuevos descendientes mediante el intercambio de los componentes de una pareja de individuos, según la probabilidad de cruce p_c para un operador de cruce dado.

Se puede observar un cambio en el procedimiento seguido al llevar a cabo el cruce según se haya especificado un solo operador de cruce o varios:

- Si solo se ha especificado un operador de cruce el procedimiento seguido es el siguiente:
  - Se genera un número aleatorio y uniformemente distribuido en el intervalo [0, 1].
  - Si $y < p_c$, se aplica el operador de cruce determinado para producir los nuevos individuos.
  - En caso contrario no se aplica cruce y los nuevos individuos generados son idénticos a sus padres.
- Si se han especificado varios operadores de cruce el procedimiento seguido es el siguiente:
  - Se genera un número aleatorio y uniformemente distribuido en el intervalo [0, 1].
  - Si $y < p_c$, se aplica el operador de cruce determinado para producir los nuevos individuos.
  - En caso contrario a y se le resta la probabilidad de cruce del operador actual, $y = y - p_c$, y se compara con la probabilidad de cruce del siguiente operador para determinar si se aplica dicho operador. Esto es así ya que las probabilidades de los distintos operadores de cruce se toman como intervalos en los que se puede encontrar y: [0, p_{c1}), [p_{c1}, $p_{c1} + p_{c2}$), ..., [$\sum_{i=1}^{n-1} p_{ci}$, $\sum_{i=1}^{n} p_{ci}$).

Este procedimiento se repite hasta que se aplique un operador de cruce o hasta que no haya más operadores de cruce. En este último caso, los nuevos individuos serán idénticos a sus padres.

En realidad, tal y como se puede observar, el procedimiento seguido cuando se ha especificado un solo operador de cruce no es más que un caso particular del procedimiento seguido cuando se han especificado varios operadores.

Los operadores de cruce disponibles son los siguientes, y se puede observar la descripción de cada uno de ellos en la sección 5.1:

- Operador de cruce de doble punto.
- Operador de cruce uniforme.
- Operador de cruce de un solo punto.

La suma de las probabilidades de los operadores de cruce debe ser mayor que 0 y menor o igual que 1.

4.2.7. Mutación

Una vez se han obtenido del paso anterior los dos nuevos individuos, tanto si se ha producido cruce como si no, cada uno de ellos se muta de acuerdo con la probabilidad de mutación, p_m, para un operador de mutación dado. En particular todas las componentes de un individuo tienen la misma probabilidad de ser mutadas, siendo exactamente p_m la probabilidad de que se produzca una mutación en dicho individuo, es decir, la probabilidad de que se mute alguna de sus componentes.

Así como ocurría en el cruce, se puede observar un cambio en el procedimiento seguido al llevar a cabo la mutación según se haya especificado un solo operador de mutación o varios:

- Si solo se ha especificado un operador de mutación el procedimiento seguido es el siguiente:
  - Para cada hijo se recorre con un bucle cada una de sus componentes.
  - Para cada componente se genera un número aleatorio z uniformemente distribuido en el intervalo [0, 1].
  - Si $z < 1 - \sqrt[n]{1 - p_m}$, se muta la componente de acuerdo a su tipo.
- Si se han especificado varios operadores de mutación el procedimiento seguido es el mismo. No obstante, si en un individuo no se ha mutado ninguna de sus componentes se repetirá el mismo proceso con dicho individuo empleando el siguiente operador de mutación.

  Este procedimiento se repite hasta que en dicho individuo se aplique un operador de mutación o hasta que no haya más operadores de mutación.

Del mismo modo, el procedimiento seguido cuando se ha especificado un solo operador de mutación no es más que un caso particular del procedimiento seguido cuando se han especificado varios operadores.

La fórmula empleada para la decisión de mutar una componente es indispensable para garantizar que la probabilidad de mutación de un individuo sea exactamente p_m y que todas sus componentes tengan la misma probabilidad de ser mutadas.

$$z \in [0, 1], \quad z < 1 - \sqrt[n]{1 - p_m} \tag{4.3}$$

.

Véase la sección §4.3 para conocer la explicación de dicha fórmula.

Los operadores de mutación disponibles para la herramienta son los siguientes, y se puede observar la descripción de cada uno de ellos en la sección 5.2:

- Operador de mutación en *1-Bit-Flip*.
- Operador de mutación en código de *Gray*.
- Operador de mutación específico para la mutación de triángulos.
- Operador de mutación de listas con probabilidad personalizada.
- Operador de mutación *Random Resetting*.
- Operador de mutación *Composite* para elegir mutación por tipos.
- Operador de mutación *Composite* para elegir mutación por posiciones.
- Operador de mutación *Creep Mutation*.

La suma de las probabilidades de los operadores de mutación debe ser mayor que 0.

4.2.8. Condiciones de terminación

El algoritmo para de realizar generaciones cuando se cumple una determinada condición de terminación. En cada generación, después de calcular la aptitud de los individuos, se comprueba si se cumple cada una de las condiciones de terminación especificadas

Las condiciones de terminación implementadas en esta herramienta son las siguientes:

- Número máximo de generaciones: El algoritmo genético termina su ejecución cuando alcanza un determinado número de generaciones.
- Porcentaje de mutantes matados: El algoritmo genético termina su ejecución cuando se ha conseguido matar un porcentaje del total de mutantes con los casos de pruebas generados.
- Todos los mutantes válidos matados: El algoritmo genético termina su ejecución cuando se ha conseguido matar todos los mutantes válidos con los casos de pruebas generados.
- Estancamiento de la aptitud máxima: El algoritmo genético termina su ejecución cuando la máxima aptitud de la población no mejora durante un determinado número de generaciones.
- Estancamiento de la aptitud promedio: El algoritmo genético termina su ejecución cuando la aptitud promedio de la población no mejora durante un determinado número de generaciones.

4.3. Fórmula de mutación por componente

El objetivo de la mutación es introducir en las poblaciones una diversidad que no se puede obtener a través del cruce. Es por ello que en la mutación se presenta el problema de encontrar la forma de poder mutar de una sola vez varias componentes de un individuo pero respetando la probabilidad de mutación efectiva del individuo, es decir, que la probabilidad de que el individuo sufra un cambio continúe siendo p_m.

Por lo tanto, hay que buscar un modo de realizar la mutación que preserve la propiedad de que la probabilidad efectiva de mutación del individuo coincida con p_m, pero que al mismo tiempo logre un óptimo reparto de la probabilidad de mutación entre las distintas componentes de un individuo.

En este sentido, la primera posibilidad que se estudió para implementar la mutación fue la siguiente:

> Para cada individuo se recorre con un bucle cada una de sus componentes. Para cada componente se genera un número aleatorio, z, uniformemente distribuido en el intervalo continuo [0, 1] y, si $z < p_m$, se muta la componente de acuerdo a su tipo.

Pero esta primera posibilidad no es del todo apropiada puesto que hace que la probabilidad efectiva de mutación de un individuo sea superior a p_m, ya que la probabilidad de mutación de cada componente del individuo es p_m. Es decir, un individuo podría sufrir cambios con una probabilidad superior a p_m en una determinada generación. Comprobémoslo con el siguiente ejemplo:

$$p_m = 0{,}1, n = 2, \text{siendo } n \text{ el número de componentes}$$
$$P(\textit{muta individuo}) = P(\textit{muta comp. 1}) + P(\textit{muta comp. 2}) - P(\textit{mutan ambas})$$
$$= 0{,}1 + 0{,}1 - 0{,}1 \cdot 0{,}1 = 0{,}19$$

Por lo que la probabilidad efectiva de mutación es casi el doble de p_m.
Para evitar este problema, se propuso la segunda posibilidad:

> Para cada individuo se recorre con un bucle cada una de sus componentes. Para cada componente se genera un número aleatorio, z, uniformemente distribuido en el intervalo continuo [0, 1] y, si $z < p_m/n$, se muta la componente de acuerdo a su tipo.

En este caso la probabilidad de mutación se reparte entre el número de componentes, n. Esto permite que se puedan mutar varias componentes de un mismo individuo en una generación sin que la probabilidad efectiva de mutación del individuo supere a p_m, ya que la probabilidad de mutación del individuo está acotada por la suma de las n probabilidades de mutación de cada componente, es decir, por $n \cdot (p_m/n) = p_m$.

El problema que presenta esta segunda posibilidad es que la probabilidad de mutación efectiva es menor que p_m, calculémosla de nuevo con los datos del ejemplo anterior:

$$0{,}05 + 0{,}05 - 0{,}05 \cdot 0{,}05 = 0{,}0975$$

Como se puede observar, no se alcanza la probabilidad indicada por p_m. Además, la diferencia aumenta ligeramente conforme crece el número de componentes o se elige un p_m mayor. Es decir, este método no es tan bueno si el número de componentes es grande. Así mismo, la probabilidad efectiva de mutación en este caso es exactamente $1-(1-p_m/n)^n$ y tiende a $1-e^{-p_m}$ conforme n crece. En la figura 4.4, se puede observar cómo decrece porcentualmente conforme varía p_m para valores suficientemente grandes de n.

La diferencia no es muy apreciable para valores bajos de p_m, pero existe y se va acentuando si se incrementa p_m, como se puede observar en la figura 4.5.

Por lo que surge la tercera posibilidad:

> Para cada individuo se recorre con un bucle cada una de sus componentes. Para cada componente se genera un número aleatorio, z, uniformemente distribuido en el intervalo continuo [0, 1] y, si $z < 1 - \sqrt[n]{1-p_m}$, se muta la componente de acuerdo a su tipo.

Con esta alternativa conseguimos que se mute cada componente del individuo con probabilidad $1 - \sqrt[n]{1-p_m}$. Esta es justo la probabilidad con la que se debe mutar cada componente para que la probabilidad efectiva de mutación del individuo sea la indicada por el parámetro p_m, teniendo además todas componentes la misma probabilidad de mutación.

Comprobémoslo tomando los datos del ejemplo anterior, ahora cada componente se mutaría con probabilidad $1 - \sqrt[2]{(1-0{,}1)} = 0{,}051\ldots$ y la probabilidad efectiva de mutación del individuo sería $0{,}051\ldots + 0{,}051\ldots - 0{,}051\ldots \cdot 0{,}051\ldots = 0{,}1$, tal y como se esperaba.

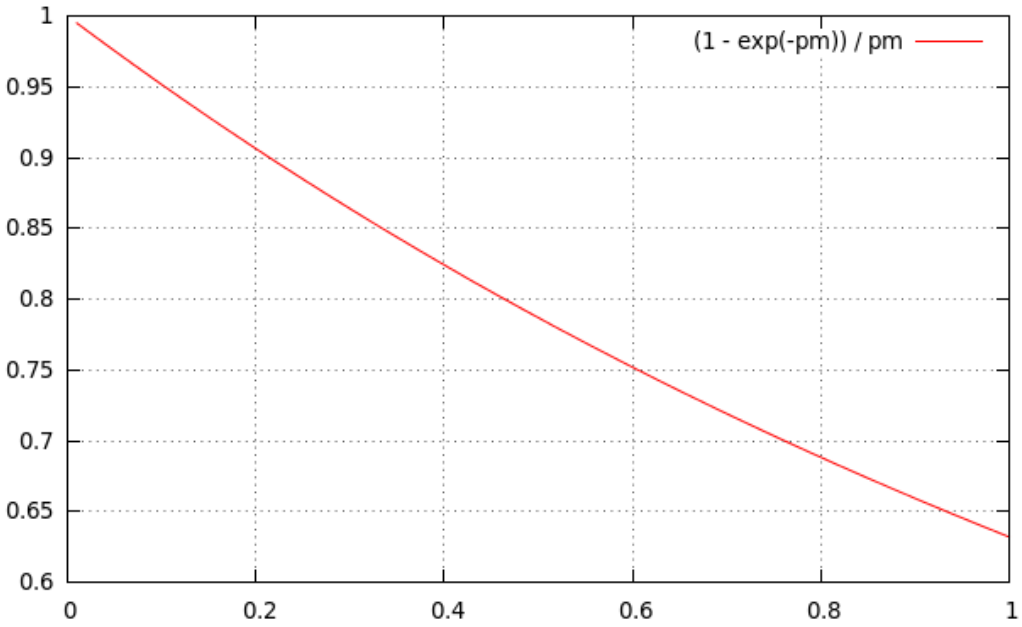


Figura 4.4: Decrecimiento conforme varía p_m para valores grandes de n.

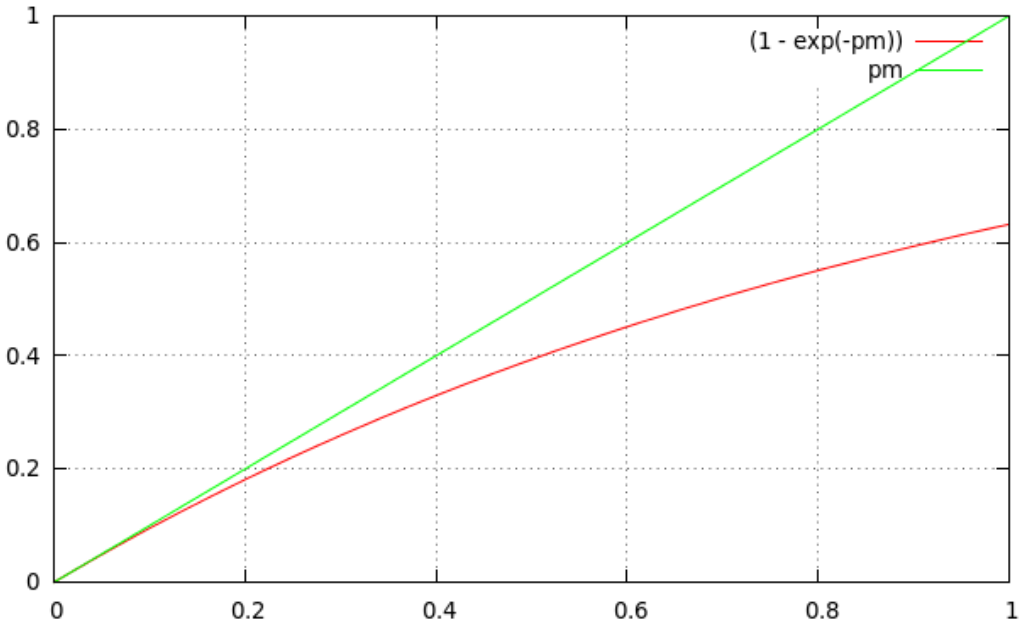


Figura 4.5: Decrecimiento para valores bajos de p_m.

Cabe recalcar que en esta tercera posibilidad el resultado global es independiente de n. Es decir, da igual lo grande o lo pequeño que sea n, todas las componentes de un individuo tienen la misma probabilidad de sufrir una mutación en cada generación y la probabilidad de que el individuo mute es exactamente p_m.

Veamos como obtener esta alternativa. Para ello se calcula la probabilidad de la unión de todos los sucesos correspondientes a mutaciones de componentes de un individuo, es decir, la probabilidad de que el individuo sufra mutación en algunas de sus componentes, que debe coincidir con p_m.

Siendo n el número de componentes de un individuo, p la probabilidad de mutación de una componente, y A, B y cada A_k, sucesos aleatorios independientes que representan la mutación de una componente concreta. Decimos que A y B son independientes si $P(A \cap B) = P(A) \cdot P(B)$. En tal caso:

$$P(A \cup B) = P(A) + P(B) - P(A) \cdot P(B)$$

Calcular la probabilidad de la unión en el caso general a partir de las fórmulas anteriores no es sencillo, incluso si nos restringimos a un número infinito de sucesos aleatorios independientes por parejas. Veamos un método alternativo:

$$P\left(\bigcup_{k<n} A_k\right) = 1 - P\left(\overline{\bigcup_{k<n} A_k}\right) = 1 - P\left(\bigcap_{k<n} \overline{A_k}\right)$$
$$= 1 - \prod_{a<k} P(\overline{A_k}) = 1 - \prod_{a<k} (1 - P(A_k))$$

En el caso especial en el que los sucesos sean equiprobables, $P(A_1) = \ldots = P(A_n) = p$ y se cumple que:

$$\prod_{a<k} (1 - P(A_k)) = (1-p)^n$$

Por lo que $P(\bigcup_{k<n} A_k) = 1 - (1-p)^n$ y $p = 1 - \sqrt[n]{1 - P(\bigcup_{k<n} A_k)}$.

Con las fórmulas resultantes fijada p podemos comprobar que la probabilidad de la unión es p_m y fijada p_m calcular qué valor debe tener p:

$$p_m = 1 - (1-p)^n$$
$$p = 1 - \sqrt[n]{1 - p_m}$$

4.4. Arquitectura para la generación de casos de prueba

En la generación de casos de prueba participan dos componentes fundamentales: el generador de mutantes y el generador de casos de prueba. En la figura 4.6 podemos ver la arquitectura diseñada para la generación de casos de prueba.

Por un lado tenemos el generador de mutantes, que es el encargado de generar a través de un programa original que recibe como entrada, todos los mutantes posibles de dicho programa original, para ello emplea los operadores de mutación disponibles para lenguaje en el que esté escrito el programa, en nuestro caso WS-BPEL 2.0. En este sentido, el grupo UCASE tiene desarrollados una serie de operadores de mutación [16] que se pueden aplicar a WS-BPEL 2.0.

Así mismo, el generador de mutantes también es empleado para ejecutar el programa original y los mutantes de éste frente a los casos de prueba, así como para obtener la matriz de ejecución que sirve para comparar las salidas producidas por el programa original y los mutantes frente a dichos casos de prueba. Dicha matriz de ejecución muestra los mutantes que han quedado vivos, muertos o inválidos y

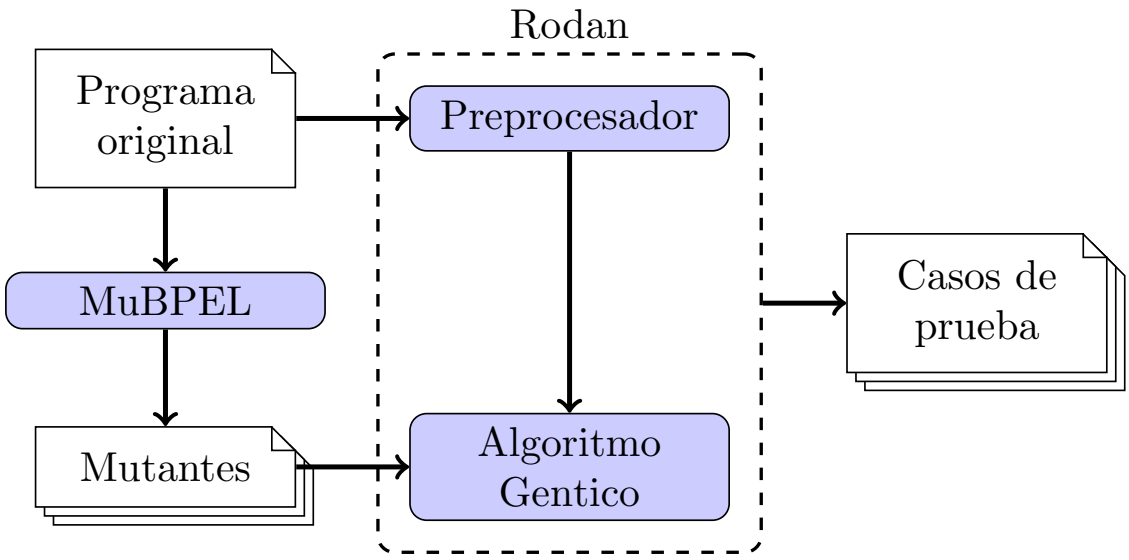


Figura 4.6: Arquitectura para la generación de casos de prueba.

es empleada por el algoritmo genético para calcular los valores de la función de aptitud para cada uno de los casos de prueba generados. En nuestro caso, el generador de mutantes empleado es la herramienta MuBPEL [17], desarrollada por el grupo UCASE para trabajar con composiciones WS-BPEL 2.0.

Por su parte, el generador de casos de prueba está dividido en dos componentes, el preprocesador y el algoritmo genético, que es en el que se centra este proyecto fin de carrera.

El preprocesador encapsula todos los aspectos dependientes del lenguaje en el que esté escrito el programa original, en este caso WS-BPEL 2.0. De esta menara, el preprocesador produce la información que necesita el algoritmo genético para que pueda producir los casos de prueba generación a generación. En este sentido, el preprocesador debe generar una especificación del formato de los datos a emplear en los casos de prueba, que se le pasa como entrada al algoritmo genético para que conozca la estructura que deben tener los casos de prueba a generar. Para ello, en primer lugar a partir de los ficheros WDSL en los que se describen las interfaces del proceso de negocio WS-BPEL 2.0 y de los servicios con los que interactúa, se produce un catálogo de mensajes que contiene las plantillas y las variables que se deben emplear en cada uno de los mensajes. A través de dicho catálogo de mensajes se puede obtener la especificación del formato que deben seguir los datos empleados en los casos de prueba. Además, esta especificación es empleada por el generador aleatorio de casos de prueba para producir un conjunto de casos de prueba que el algoritmo genético usará como población inicial.

A su vez, el algoritmo genético recibe la especificación de la estructura que deben seguir los casos de prueba y un conjunto de casos de prueba que constituirá la población inicial del algoritmo genético. A partir de esta población inicial el algoritmo genético producirá, generación a generación, nuevos casos de prueba con el objetivo de matar el mayor número posible de mutantes del programa original, de acuerdo al procedimiento que se ha explicado anteriormente.

El algoritmo genético emplea un fichero de configuración YAML [18] en el que se indican todos los aspectos que el algoritmo necesita para su ejecución. En el listado 4.2 se puede observar un ejemplo de dicho fichero de configuración. En él se detallan:

- El tamaño que debe tener la población en cada una de las generaciones realizadas por el algoritmo genético.
- La semilla que se debe emplear para la generación aleatoria durante la ejecución del algoritmo genético.

- El *executor* que representa al generador de mutantes empleado por el algoritmo genético para crear los mutantes del programa original y ejecutarlos frente a los casos de prueba producidos en cada generación. Además se indican las rutas del fichero de casos de prueba basado en plantillas empleado, el programa en WS-BPEL 2.0 original y el fichero de salida, así como el número de hilos en los que dividir la ejecución.
- El *parser* empleado para obtener los tipos de las variables indicadas en el fichero de especificación que contiene la estructura de los casos de prueba a emplear.
- El *formatter* que indica el formato que deben seguir los datos.
- La estrategia empleada para la generación aleatoria de nuevos datos.
- Los operadores de cruce a emplear, con su probabilidad de aplicación indicada.
- Los operadores de mutación a emplear, con su probabilidad de aplicación indicada.
- Los generadores de nuevos individuos, con su probabilidad de aplicación indicada.
- Los operadores de selección, con su probabilidad de aplicación indicada.
- Las condiciones de terminación que debe tener en cuenta el algoritmo genético al finalizar cada generación.
- Los *loggers* que representan los informes que debe generar el algoritmo genético.
- Los operadores de mutación del lenguaje del programa original que no se desean utilizar. Así como también es posible indicar los mutantes específicos con los que se desea trabajar.

Listado 4.2: Ejemplo de fichero de configuración

```
1  populationSize: 50
2
3  seed: "7901344213630061"
4
5  # On a quad-core i7 machine (with HT), this should leave 1 core free for
6  # doing other things (such as tending to SSH sessions).
7  executor: !!gamera.exec.bpel.BPELExecutor
8    testSuite: Triangle/suite-templates.bpts
9    originalProgram: Triangle/Triangle.bpel
10   outputFile: process.output
11   comparisonThreads: 4
12
13 parser: !!testgen.parsers.spec.xtext.integration.XtextSpecParser {spec:
       Triangle/data.spec}
14 formatter: &f !!testgen.formatters.VelocityFormatter {}
15 strategy: !!testgen.strategies.random.UniformRandomStrategy {}
16
17 crossoverOperators:
18   !!gamera.ggen.genetic.crossover.ComponentCrossoverOperator {} : {
         probability: 0.9}
19
20 mutationOperators:
21   !!gamera.ggen.genetic.mutation.BinaryMutationOperator {mutationRange: 20}
         : {probability: 0.1}
```

```
22
23 individualGenerators:
24   !!gamera.ggen.generate.RandomGenerator {} : {percent: 0.1}
25
26 selectionOperators:
27   !!gamera.ggen.select.RouletteSelection  {} : {percent: 1}
28
29 terminationConditions:
30   - !!gamera.ggen.term.GenerationCountCondition   {count: 100}
31   - !!gamera.ggen.term.StagnationAverageFitness   {count: 8,
       relativeMinimumChange: 0.01}
32   - !!gamera.ggen.term.AllValidProgramMutantsKilledCondition {}
33
34 loggers:
35   - !!gamera.ggen.log.MessageLogger {console: false, file: gamerahom-ggen-
       messages-1.log, verbose: true}
36   - !!gamera.ggen.log.FullHistoryReportLogger {file: history-1.txt}
37   - !!gamera.ggen.log.UniqueByKillsGenerationReportLogger {file:
       uniqueByKills-1.txt}
38   - !!gamera.ggen.log.LastPopulationDataLogger {file: lastPopulation-1.vm,
       formatter: *f}
39   - !!gamera.ggen.log.UniqueByComponentsDataLogger {file:
       uniqueByComponents-1.vm, formatter: *f}
40   - !!gamera.ggen.log.UniqueByKillsDataLogger {file: uniqueByKills-1.vm,
       formatter: *f}
41   - !!gamera.ggen.log.SubsumeByKillsDataLogger {file: uniqueByKillsSubsumed
       -1.vm, formatter: *f}
42   - !!gamera.ggen.log.UniqueByComponentsGenerationReportLogger {file:
       individualsHistory-1.txt}
43   - !!gamera.ggen.log.HTMLDataLogger {file: Triangle_Trace-1.html}
44
45 excludedOperators: cco, cde
46
47 # Removing killed program mutants is disabled
48 removeKilledBy: 0
```

Capítulo 5

Operadores genéticos

En este capítulo se detallan todos los operadores de cruce y mutación de los que dispone el sistema.

5.1. Operadores de cruce

Los operadores de cruce de los que dispone el sistema siguen la estructura que se puede observar en la figura 5.1. En concreto, todos los operadores de cruce desarrollados implementan la interfaz *GACrossoverOperator*.

5.1.1. Operador de cruce de doble punto

Con este operador de cruce se generan dos nuevos individuos a partir del cruce de componentes entre dos puntos de dos individuos dados.

En concreto, dados dos individuos con un número de componentes n, se generan dos enteros aleatorios p y q uniformemente distribuidos en el intervalo [0, n+1). Donde p y q representan los puntos de cruce que se van a emplear para realizar el mismo, es decir, p es la posición del componente a partir del cual se realiza el intercambio de componentes entre el par de individuos y q es la posición donde termina de hacerse el cruce.

En la figura 5.2 podemos ver un ejemplo ilustrativo del comportamiento de este operador:

Cabe destacar que todos los individuos deben tener el mismo número de componentes y que no se puede realizar un cruce entre individuos que solo estén constituidos por un componente. Además, se fuerza a que p sea menor que q y se garantiza que p no sea igual a q, ya que no se produciría intercambio de componentes, ni que p sea 0 y q sea n+1, ya que lo que produciría un cruce con estos dos puntos es el

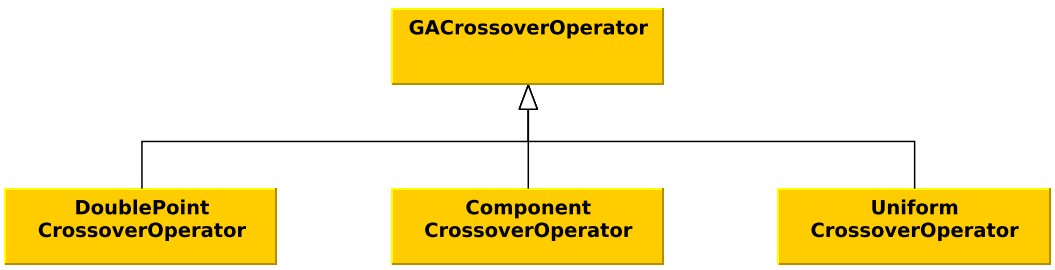


Figura 5.1: Estructura de los operadores de cruce.

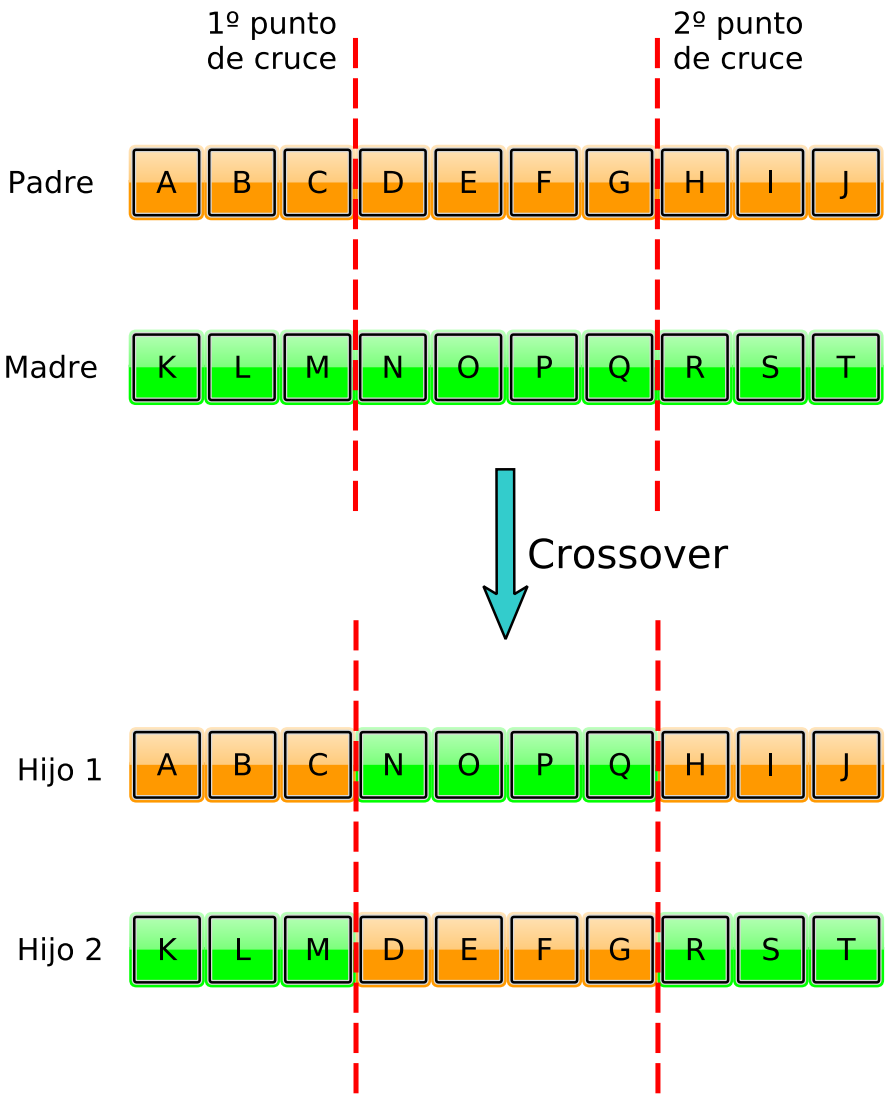


Figura 5.2: Comportamiento del operador de cruce de doble punto.

intercambio de todos los componentes de los individuos, es decir, se producirían dos individuos iguales a los padres.

5.1.2. Operador de cruce uniforme

Con este operador de cruce se generan dos nuevos individuos a partir del cruce aleatorio de entre dos individuos dados.

En concreto, dados dos individuos con un número de componentes n:

- Se recorre cada uno de las componentes y para cada uno de ellos se genera un booleano aleatorio b uniformemente distribuido en el intervalo [0, 1].
- Si el booleano generado es verdadero se realiza el cruce de la componente correspondiente entre los dos individuos.
- En caso contrario no se realiza el cruce de la componente actual y se sigue realizando el mismo proceso para el resto de componentes.

De esta manera, se consigue que cada componente de los individuos se cruce con una probabilidad del 0.5, es decir, el equivalente a lanzar una moneda para decidir si una determinada componente se cruza o no.

En la figura 5.3 podemos ver un ejemplo ilustrativo del comportamiento de este operador:

Cabe destacar que todos los individuos deben tener el mismo número de componentes. Así mismo, no se puede realizar un cruce entre individuos que solo estén constituidos por un componente, ya que el resultado sería dos individuos exactamente iguales a los padres, puesto lo que se realizaría es un intercambio de componentes entre individuos.

5.1.3. Operador de cruce de un solo punto

En este operador de cruce dados dos individuos con un número de componentes n, se genera un entero aleatorio, p, uniformemente distribuido en el intervalo [1, n). p representa el punto de cruce, es decir, la posición del componente a partir del cual se realiza el intercambio de componentes entre el par de individuos. Cabe destacar que todos los individuos deben tener el mismo número de componentes, además, no se puede realizar cruce de individuos que solo estén constituidos por un componente.

5.2. Operadores de mutación

En la figura 5.4 se ilustra la estructura que siguen los operadores de mutación desarrollados en el sistema.

Como podemos observar, los operadores de mutación específico para la mutación de triángulos, el operador *Creep Mutation* y el operador abstracto *Composite* implementan la interfaz *GAMutationOperator*. Mientras que el operador de mutación *1-Bit Flip*, el operador de mutación en código de Gray, el operador de mutación de listas con probabilidad personalizada y el operador *Random Resetting* extienden al operador de mutación *Creep Mutation*. Por otro lado, los operadores de mutación *Composite* para elegir mutación por tipos y por posiciones, extienden al operador abstracto *Composite*.

5.2.1. Operador de mutación *1-Bit Flip*

Este operador de mutación modifica directamente la representación interna de un número, expresado en codificación binaria. Para ello se selecciona un bit al azar para negarlo y así alterar el valor original.

En concreto, el operador se comporta para la mutación de enteros de la siguiente manera:

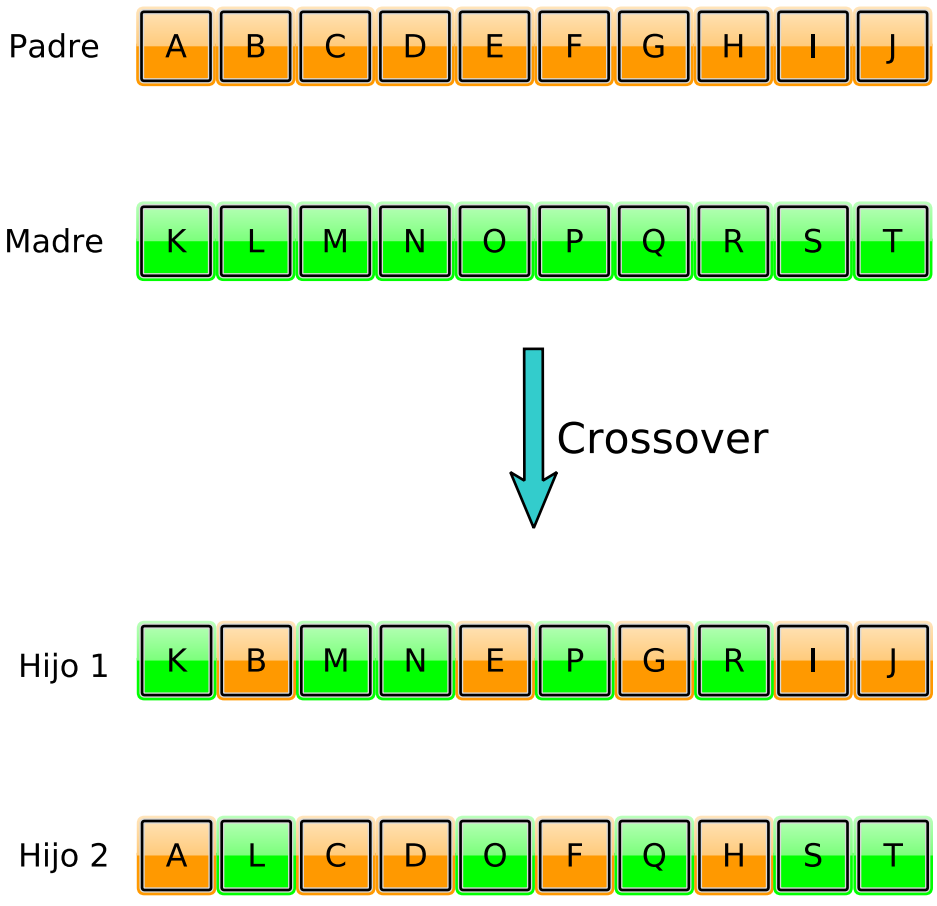


Figura 5.3: Comportamiento del operador de cruce uniforme.

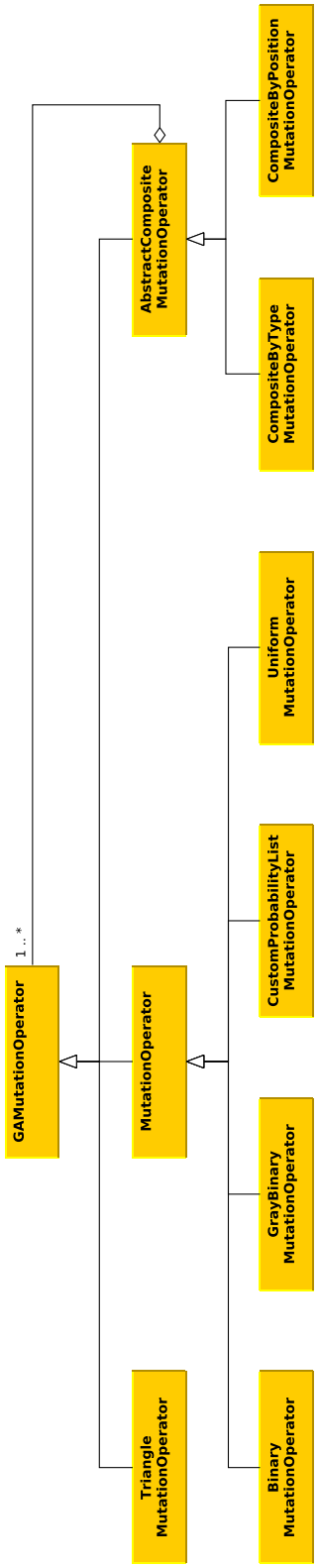


Figura 5.4: Estructura de los operadores de mutación.

- En primer lugar se resta el valor mínimo permitido al valor máximo permitido, $|max - min|$. El número n de bits necesario para representar el valor absoluto resultante de dicha resta será el rango que se utilice para elegir el bit a mutar.
- Se genera un entero aleatorio uniformemente distribuido en el intervalo [0, n). Dicho entero representará el índice del bit elegido para cambiar su valor.
- A continuación se resta al valor original x el valor mínimo permitido. Por lo que el valor a mutar es $x - min$. Con ello, se desplaza el valor a muta del rango $[min, max]$ al rango $[0, max - min]$.
- Al nuevo valor a mutar se le modifica el bit seleccionado previamente.
- Al valor resultante de la mutación se le suma el valor mínimo permitido para obtener el resultado final de la mutación. De esta manera, el resultado vuelve a pasar del rango $[0, max - min]$ al rango $[min, max]$.
- Por último, si el valor mutado resultante es mayor que el valor máximo permitido, el valor mutado toma el valor máximo permitido.

Podemos ver un ejemplo detallado del comportamiento de este operador en la figura 5.5.

Como se puede observar en el comportamiento descrito anteriormente, se consigue evitar el problema de que el valor resultante de la mutación sea menor que el valor mínimo permitido puesto que al realizar la operación $x - min$, garantizamos que el nuevo valor obtenido se sitúe dentro del rango $[0, max - min]$. Por lo que el menor valor que puede resultar de una mutación es 0, que se convertiría en min posteriormente al aplicar la operación $x + min$ para movernos de nuevo al rango $[min, max]$. Este procedimiento es indiferente al signo de los valores con los que se realice la mutación, es decir, es válido tanto para números positivos como para negativos.

Esta forma de operar facilita además la mutación teniendo en cuenta que max y min pueden ser de distinto signo. De otra forma, se presentarían algunos problemas que con la implementación actual quedan resueltos. Si por ejemplo max es positivo, min es negativo y en la mutación se tiene en cuenta el cambio de signo, el valor absoluto de max y min puede ser distinto, por lo que el nº de bits necesarios para representar a cada uno de ellos también podría ser distinto, así que según el signo que tuviera el valor a mutar habría que decidir en cual de los dos rangos de valores moverse para elegir un bit al azar para realizar la mutación. Esto dificultaría además la realización de un cambio de signo, puesto que una posible alta diferencia de tamaño entre los valores absolutos de max y min podría provocar que al pasar de un número entero positivo a uno negativo, el valor resultante quedase por debajo de min, o viceversa. Por lo tanto, si la diferencia entre los valores absolutos de max y min es alta, los valores resultantes de un cambio de signo podrían tender a tomar el valor max o min respectivamente.

Dicho problema queda resuelto con el comportamiento implementado, puesto que se consigue reducir el dominio de valores posibles al rango específico $[0, max - min]$, mediante la operación $x - min$. Es decir, dentro de ese rango queda enmarcado también el cambio de signo durante la mutación. Al aplicar después $x + min$, el nuevo valor queda comprendido dentro del rango de valores original, $[min, max]$. Puede ocurrir que la mutación dé como resultado un valor mayor a max, en ese caso se recortaría al valor max correspondiente.

Cabe destacar que cuando $|max - min|$ no es cercano a $2^n - 1$ se puede producir clipping por el lado del máximo, lo que podría producir un número considerable de mutaciones que dan como resultado el valor máximo permitido del tipo a mutar. Por lo que puede ser bastante más útil emplear este operador para tipos en los que el resultado de $|max - min|$ es cercano a $2^n - 1$.

Para la mutación de flotantes se presenta la disyuntiva de elegir la forma de matación más adecuada para poder mutar la parte entera y/o la parte decimal. En este operador de mutación ese problema se resuelve mutando un valor no escalado del número en coma flotante, es decir, se convierte el valor a

Figura 5.5: Ejemplo de comportamiento del operador de mutación binario.

mutar y el máximo y mínimo permitidos a números enteros para poder realizar la mutación empleando el método implementado para mutar números enteros. De esta manera, se resuelve dicho problema, ya que se muta el valor entero, que incluye la parte entera y decimal del número en coma flotante, y luego se vuelve a convertir el entero resultante de la mutación en un número en coma flotante teniendo en cuenta la escala que tenía originalmente.

En la mutación de flotantes, el operador se comporta de la siguiente manera en concreto:

- Dados el valor a mutar, x, el valor máximo permitido, max, y el valor mínimo permitido, min. Se obtiene la escala máxima de entre estos tres valores.

- Se mueve el punto decimal de cada valor mencionado anteriormente tantas veces a la derecha como la escala indique. Es decir, se convierten los tres valores de flotante a entero, siendo su valor, respectivamente:

  - $x = x * 10^{escala}$
  - $min = min * 10^{escala}$
  - $max = max * 10^{escala}$

  De esta manera, se consigue mutar los valores en coma flotante respetando, de una manera sencilla, los valores máximos y mínimos entre los que debe estar enmarcado el valor resultante, puesto que todos los valores se multiplican por la misma escala.

- A continuación, se muta el valor empleando la mutación binaria implementada para números enteros.

- Por último, el valor mutado se vuelve a escalar empleando la escala calculada anteriormente. Es decir, el valor resultante es:

  $r = r * 10^{-escala}$

5.2.2. Operador de mutación en codificación de Gray

En este operador se realiza la mutación empleando la representación binaria de Gray de un número, modificando su representación interna mediante la negación de un bit elegido aleatoriamente.

Este operador se ha implementado en comparación con el operador de mutación *1-Bit Flip*, puesto que con la codificación de Gray se pueden evitar los problemas relacionados con la llamada distancia de *Hamming*. La distancia de *Hamming* es el número de bits en el que difieren dos números en su representación binaria, es decir, es el número de bits que hay que voltear para pasar de un número a otro.

Empleando una representación binaria en codificación de Gray, se puede evitar la aparición de los llamados *Hamming cliffs*, que se trata de un efecto que se puede producir porque dos números próximos entre sí pueden estar a una distancia de *Hamming* relativamente grande. En la figura 5.6, podemos ver un ejemplo ilustrativo sobre la distancia *Hamming* y los *Hamming cliffs* que se producen entre dos números cercanos entre sí.

En codificación de Gray, dos números consecutivos difieren en un único bit, por eso su distancia de *Hamming* es 1.

Cuanto mayor es la distancia de *Hamming* entre dos números, más difícil es que se pueda producir un número a partir del otro, ya que se requiere el cambio de un mayor número de bits para poder conseguirlo. Esta circunstancia puede resultar un inconveniente para la resolución de algunos problemas, por lo que se hace útil la implementación de un operador que utilice la codificación binaria de Gray.

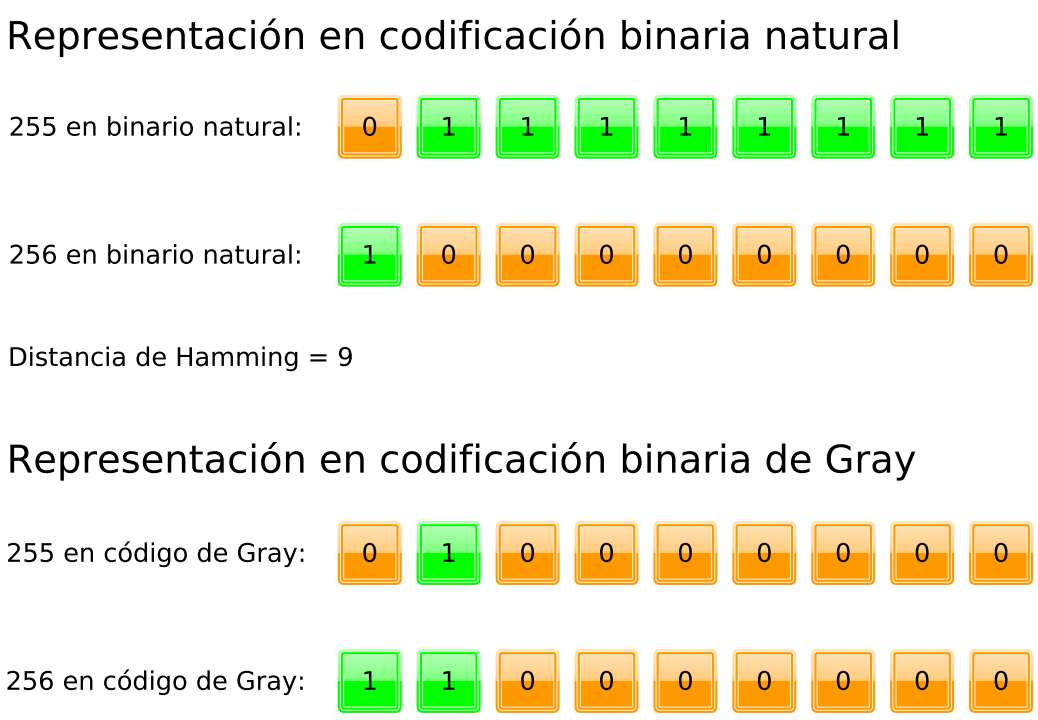


Figura 5.6: Ejemplo sobre la distancia de Hamming entre dos números próximos.

Algoritmo 2 Conversión de codificación de Gray a binario natural

1: {x es el número a convertir.}
2: **mientras** $x \gg 1 \neq 0$ **hacer**
3: $\quad x = x \oplus (x \gg 1)$
4: **fin mientras**

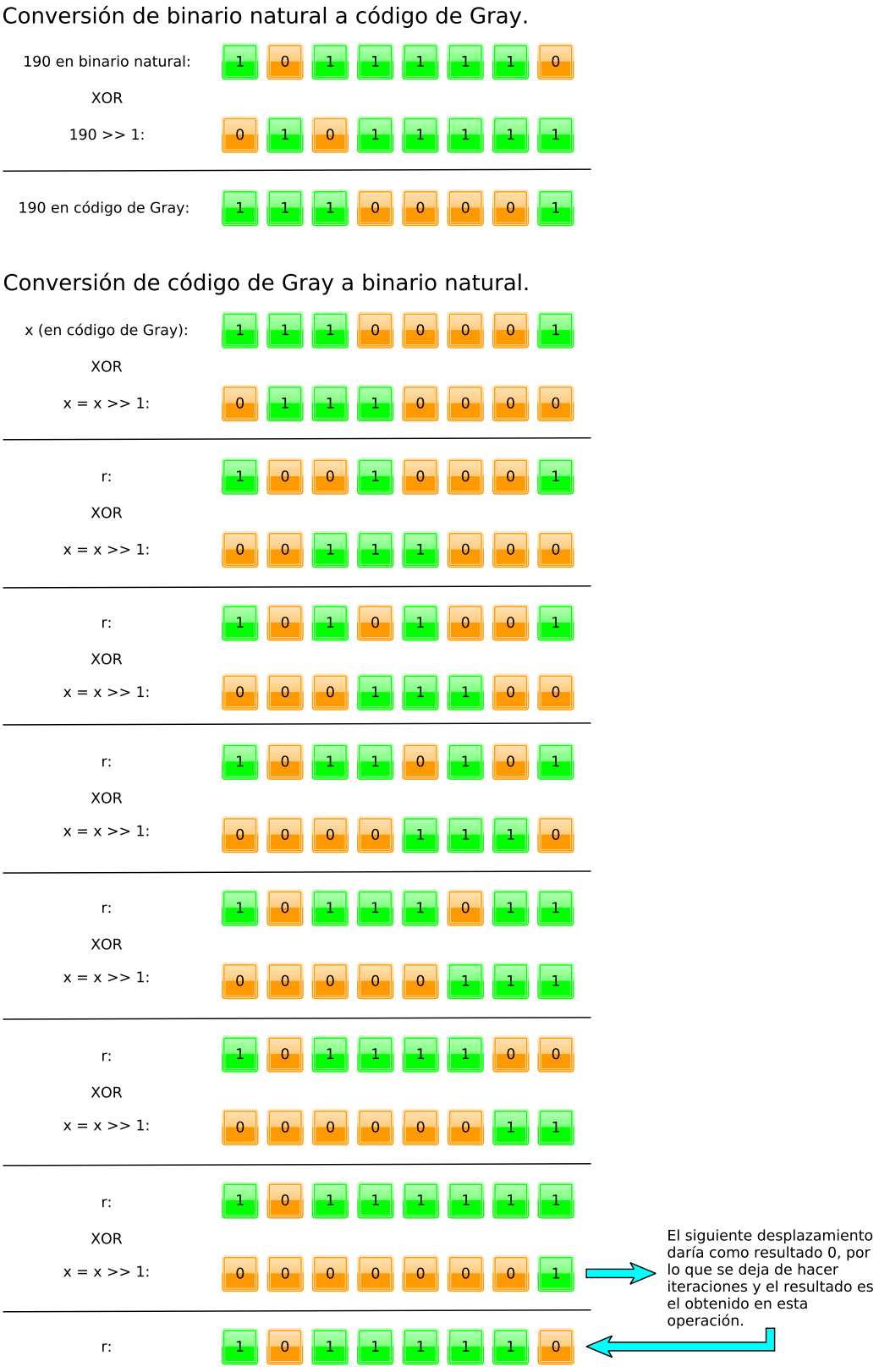


Figura 5.7: Ejemplo de conversiones de binario a Gray y viceversa.

Para convertir un número natural en binario natural a codificación de Gray es necesario hacer un XOR bit a bit entre x y su desplazamiento a la derecha. Así mismo, para convertir un número de codificación de Gray a binario natural es necesario seguir el procedimiento indicado en el algoritmo 2. Podemos ver un ejemplo ilustrativo del funcionamiento de ambas conversiones en la figura 5.7.

El comportamiento de este operador se puede definir de la siguiente manera:

- En primer lugar se resta el valor mínimo permitido al valor máximo permitido, $|max - min|$. El número n de bits necesario para representar el valor absoluto resultante de dicha resta será el rango que se utilice para elegir el bit a mutar.
- Se genera un entero aleatorio uniformemente distribuido en el intervalo [0, n). Dicho entero representará el índice del bit elegido para cambiar su valor.
- A continuación se resta al valor original x el valor mínimo permitido. Por lo que el valor a mutar es $x - min$. Con ello, se desplaza el valor a mutar del rango $[min, max]$ al rango $[0, max - min]$.
- El valor a mutar se convierte a codificación de Gray.
- Al valor en codificación de Gray se le modifica el bit seleccionado previamente.
- Se convierte de nuevo el valor mutado de codificación de Gray a binario.
- Al valor resultante de la mutación se le suma el valor mínimo permitido para obtener el resultado final de la mutación. De esta manera, el resultado vuelve a pasar del rango $[0, max - min]$ al rango $[min, max]$.
- Por último, si el valor mutado resultante es mayor que el valor máximo permitido, el valor mutado toma el valor máximo permitido.

Al igual que en el operador *1-Bit Flip*, se consigue evitar el problema de que el valor resultante de la mutación sea menor que el valor mínimo permitido puesto que al realizar la operación $x - min$, garantizamos que el nuevo valor obtenido se sitúe dentro del rango $[0, max - min]$. Por lo que el menor valor que puede resultar de una mutación es 0, que se convertiría en min posteriormente al aplicar la operación $x + min$ para movernos de nuevo al rango $[min, max]$. Este procedimiento es indiferente al signo de los valores con los que se realice la mutación, es decir, es válido tanto para números positivos como para negativos.

Esta forma de operar facilita además la mutación teniendo en cuenta que max y min pueden ser de distinto signo. De otra forma, se presentarían algunos problemas que con la implementación actual quedan resueltos. Si por ejemplo max es positivo, min es negativo y en la mutación se tiene en cuenta el cambio de signo, el valor absoluto de max y min puede ser distinto, por lo que el nº de bits necesarios para representar a cada uno de ellos también podría ser distinto, así que según el signo que tuviera el valor a mutar habría que decidir en cual de los dos rangos de valores moverse para elegir un bit al azar para realizar la mutación. Esto dificultaría además la realización de un cambio de signo, puesto que una posible alta diferencia de tamaño entre los valores absolutos de max y min podría provocar que al pasar de un número entero positivo a uno negativo, el valor resultante quedase por debajo de min, o viceversa. Por lo tanto, si la diferencia entre los valores absolutos de max y min es alta, los valores resultantes de un cambio de signo podrían tender a tomar el valor max o min respectivamente.

Dicho problema queda resuelto con el comportamiento implementado, puesto que se consigue reducir el dominio de valores posibles al rango específico $[0, max - min]$, mediante la operación $x - min$. Es decir, dentro de ese rango queda enmarcado también el cambio de signo durante la mutación. Al aplicar después $x + min$, el nuevo valor queda comprendido dentro del rango de valores original, $[min, max]$. Puede ocurrir que la mutación dé como resultado un valor mayor a max, en ese caso se recortaría al valor max correspondiente.

Cabe destacar que cuando $|max - min|$ no es cercano a $2^n - 1$ se puede producir clipping por el lado del máximo, lo que podría producir un número considerable de mutaciones que dan como resultado el valor máximo permitido del tipo a mutar. Por lo que puede ser bastante más útil emplear este operador para tipos en los que el resultado de $|max - min|$ es cercano a $2^n - 1$.

Para la mutación de flotantes se presenta la disyuntiva de elegir la forma de mutación más adecuada para poder mutar la parte entera y/o la parte decimal. En este operador de mutación ese problema se resuelve mutando un valor no escalado del número en coma flotante, es decir, se convierte el valor a mutar y el máximo y mínimo permitidos a números enteros para poder realizar la mutación empleando el método implementado para mutar números enteros. De esta manera, se resuelve dicho problema, ya que se muta el valor entero, que incluye la parte entera y decimal del número en coma flotante, y luego se vuelve a convertir el entero resultante de la mutación en un número en coma flotante teniendo en cuenta la escala que tenía originalmente.

En la mutación de flotantes, el operador se comporta de la siguiente manera en concreto:

- Dados el valor a mutar, x, el valor máximo permitido, max, y el valor mínimo permitido, min. Se obtiene la escala máxima de entre estos tres valores.
- Se mueve el punto decimal de cada valor mencionado anteriormente tantas veces a la derecha como la escala indique. Es decir, se convierten los tres valores de flotante a entero, siendo su valor, respectivamente:
  - $x = x * 10^{escala}$
  - $min = min * 10^{escala}$
  - $max = max * 10^{escala}$

  De esta manera, se consigue mutar los valores en coma flotante respetando, de una manera sencilla, los valores máximos y mínimos entre los que debe estar enmarcado el valor resultante, puesto que todos los valores se multiplican por la misma escala.
- A continuación, se muta el valor empleando la mutación binaria implementada para números enteros.
- Por último, el valor mutado se vuelve a escalar empleando la escala calculada anteriormente. Es decir, el valor resultante es:

  $r = r * 10^{-escala}$

5.2.3. Operador de mutación TriangleMutationOperator

El objetivo de este operador de mutación es conseguir modificar los lados de un triángulo de una determinada manera. Se ha implementado con el objetivo de tener un operador específico para la composición Triangle, una composición que resuelve el problema de clasificación de triángulos según el tamaño de sus lados.

La composición Triangle clasifica un triángulo atendiendo a la longitud de cada uno de sus lados. Cada resultado que proporciona la composición Triangle para la clasificación de los distintos tipos de triángulos, depende del cumplimiento una de las siguientes características:

- Alguno de los lados no es mayor que 0.
- Alguno de los lados es mayor que la suma de los otros dos.
- Los tres lados son iguales.

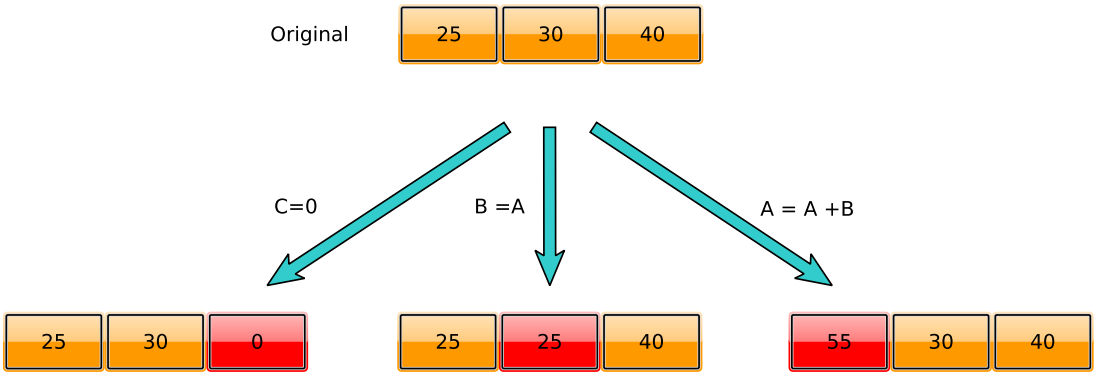


Figura 5.8: Mutación con el operador TriangleMutationOperator.

- Dos de los lados son iguales.
- Todos los lados tienen una longitud distinta.

Así que, un operador ideal para este problema debe poder cambiar de un modo relativamente sencillo de un tipo de triángulo a otro. Esto se puede conseguir con las posibles mutaciones que se han implementado en este operador para la mutación de una componente concreta de un triángulo. Son las siguientes:

- Establecer a 0 el valor de la componente a mutar.
- Copiar el valor de otra componente a la componente a mutar.
- Sumar al valor de la componente a mutar el valor de otra componente.

Para la selección de la componente origen a copiar o sumar, se genera un entero aleatorio z uniformemente distribuido en el intervalo [0, 3), que indicará el índice de dicha componente.

En la figura 5.8 podemos ver en detalle las posible mutaciones que se pueden producir mediante el uso de este operador.

De esta manera, se ha implementado este operador específico para la resolución de este problema, de tal modo que se pueda considerar como un operador ideal para el mismo y que se pueda tomar como referencia en comparación con otros operadores de mutación para la resolución de este problema.

5.2.4. Operador de mutación de listas con probabilidad personalizada

Este operador de mutación se ha desarrollado para disponer de un nuevo método de mutación de listas. De esta manera, en este operador se ha implementado un comportamiento similar al que se emplea en el algoritmo genético para decidir las componentes a mutar de un determinado individuo. En este sentido, se recorren los n elementos de la lista a mutar decidiendo si toca o no mutar cada uno de ellos, de forma que todos los elementos posean la misma probabilidad de ser mutados, $p = 1 - \sqrt[n]{1 - p_{me}}$.

Siendo p_{me} la probabilidad de que se mute algún elemento de la lista. Además, en este operador se tiene la opción de eliminar uno de sus elementos de acuerdo a una probabilidad prefijada p_e, así como de insertar un elemento de acuerdo a otra probabilidad prefijada p_i.

Cabe destacar que en cada mutación solo se puede producir uno de los tres tipos posibles (modificación, eliminación e inserción), ya que son tres sucesos mutuamente excluyentes. De esta manera, la suma de las probabilidades de los tres debe ser 1, $p_{me} + p_i + p_e = 1$.

Podemos ver un ejemplo detallado del comportamiento de este operador en la figura 5.9. En concreto, el operador se comporta de la siguiente manera:

- Si la longitud de la lista no es mayor que el número mínimo de elementos, no se pueden eliminar elementos y $p_e = 0$, independientemente del valor prefijado que tuviese con anterioridad.
- Si la longitud de la lista no es menor que el número máximo de elementos, no se pueden eliminar elementos y $p_i = 0$, independientemente del valor prefijado que tuviese con anterioridad.
- La probabilidad de modificar un elemento es $p_{me} = 1 - p_i - p_e$, excepto si p_i y p_e son iguales a 0, lo que supondría la mutación de todos los elementos de la lista. En este caso, p_{me} tomaría el valor 0.999 por defecto para reducir la probabilidad de que cada elemento de la lista pueda ser mutado. Cabe destacar que el valor que toma p_{me} en este caso es modificable por el usuario.
- Se recorren los elementos de la lista y para cada uno de ellos se genera un número aleatorio z uniformemente distribuido en el intervalo continuo [0,1]. Si $z < 1 - \sqrt[n]{1 - p_{me}}$ se muta el elemento correspondiente de acuerdo a su tipo.
- Si se ha mutado algún elemento se ha terminado el proceso de mutación de la lista.
- En caso contrario, si $p_e > 0$ o $p_i > 0$, se genera un número aleatorio x uniformemente distribuido en el intervalo continuo $[0, p_e + p_i)$.
- Si $x < p_e$, se elige un elemento aleatoriamente de la lista y se elimina.
- En caso contrario, se elige una posición aleatoriamente de la lista y se inserta un elemento.

Este proceso se repite hasta que ha tenido lugar algún tipo de mutación en la lista, ya que existe la posibilidad de que si la probabilidad de eliminación e inserción son 0, no se modifique ningún elemento de la lista en un primer barrido. Por lo que es necesario continuar hasta que se haya realizado alguna mutación, ya que la probabilidad de que la lista mute debe ser 1.

5.2.5. Patrón *Composite*

El objetivo del patrón *Composite* es la facilidad de uso y en nuestro caso, permitir la elección de diferentes operadores de mutación de acuerdo a un determinado criterio. Este patrón permite la construcción de objetos complejos a través de otros más simples y que además son semejantes entre ellos, debido a la composición recursiva y a la estructura en forma de árbol que presenta. De esta manera, posibilita la creación de objetos complejos a través de componer recursivamente en una estructura de árbol objetos que implementen a una misma interfaz.

En la figura 5.10 podemos ver la estructura seguida por el patrón *Composite*.

Como se puede observar, se dispone de una interfaz común *GAMutationOperator* que determina el molde que deben seguir los operadores de mutación. A su vez, tanto los operadores de mutación implementados, como la clase que define el patrón *Composite* implementan la interfaz *GAMutationOperator*. Además, la clase *Composite* establece una relación todo-parte con la interfaz que implementa, consiguiendo así que cada *Composite* puede contener operadores de mutación y otros *Composite* que a su vez pueden contener más objetos recursivamente.

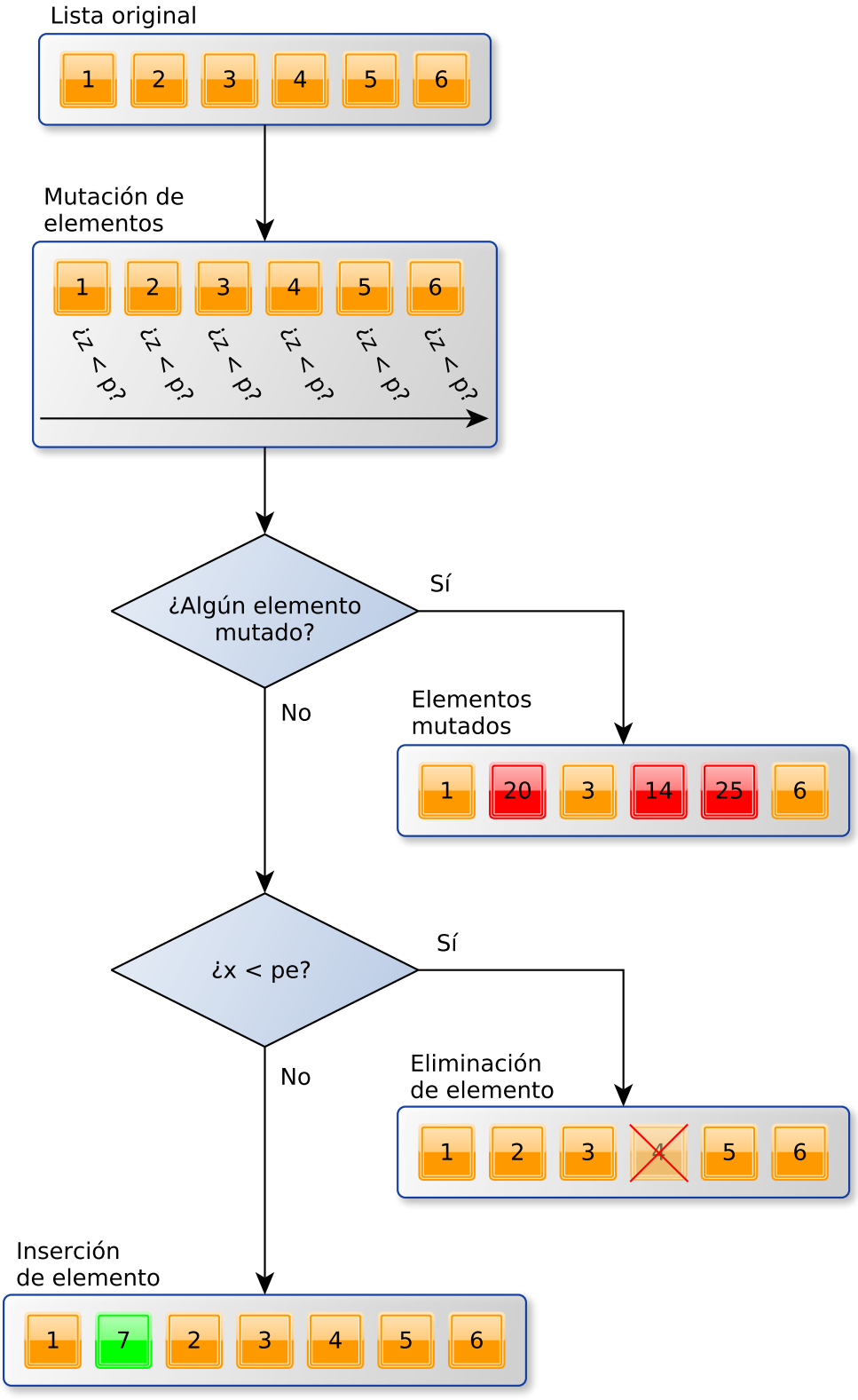


Figura 5.9: Ejemplo de comportamiento del operador de mutación de listas con probabilidad personalizada.

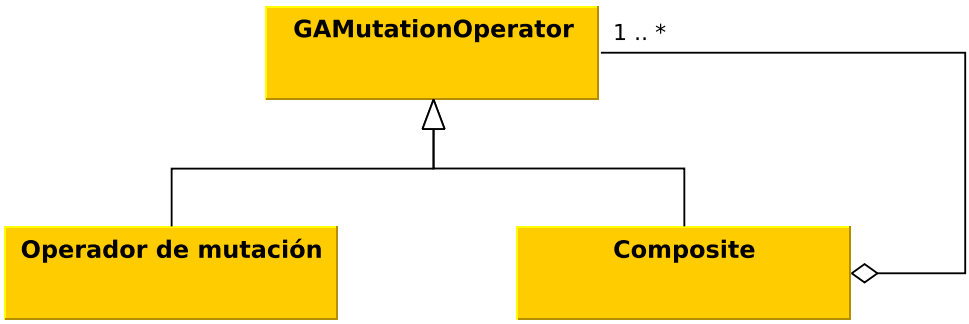


Figura 5.10: Estructura seguida por el patrón *Composite*.

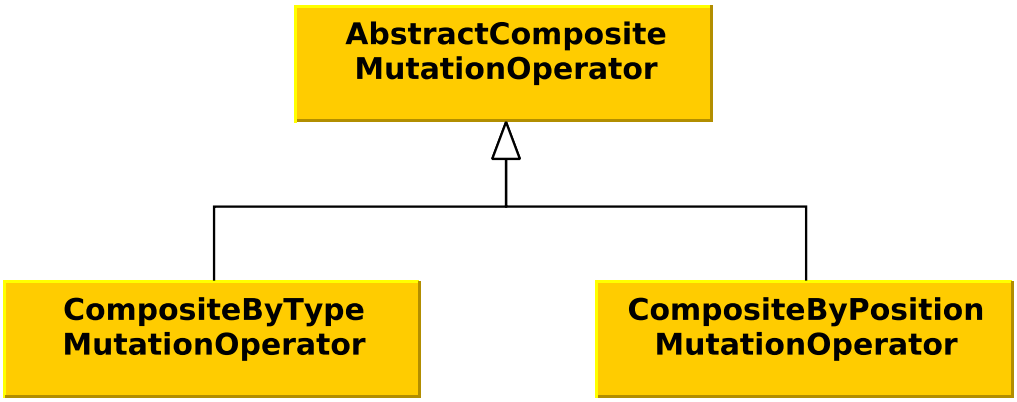


Figura 5.11: Implementación del patrón *Composite*.

De esta manera, se facilita la manipulación tanto de los objetos simples como de los compuestos, ya que ambos implementan una interfaz común y, por lo tanto, se tratan de manera uniforme.

La clase *Composite* implementada en este sistema se trata de una clase abstracta *AbstractCompositeMutationOperator* que es extendida por las clases *CompositeByTypeMutationOperator* y *CompositeByPositionMutationOperator*, como podemos observar en la figura 5.11. Estas dos clases representan respectivamente al operador de mutación *Composite* para elegir mutación por tipos y al operador de mutación *Composite* para elegir mutación por posiciones.

5.2.6. Operador de mutación *Composite* para elegir mutación por tipos

Este operador de mutación permite especificar qué operadores de mutación deben emplearse para cada uno de los distintos tipos de datos que presenten las componentes que conforman los individuos de la población.

En la figura 5.12 podemos observar un ejemplo ilustrativo.

Gracias a este operador no se tiene que emplear un solo operador de mutación para todos los tipos de datos, lo que limitaría mucho el marco de actuación a la hora de realizar estudios de investigación. De esta manera, se pueden combinar todos los operadores de mutación que se deseen de acuerdo a los tipos de datos de los componentes de los individuos.

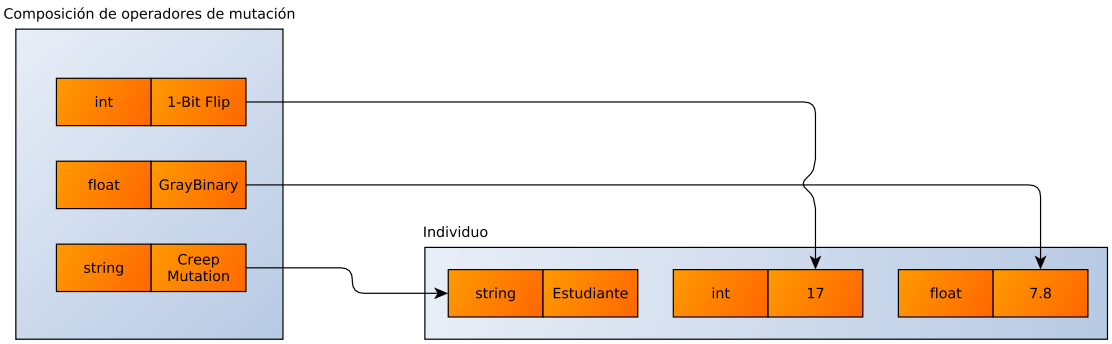


Figura 5.12: Operador de mutación *Composite* para elegir mutación por tipos.

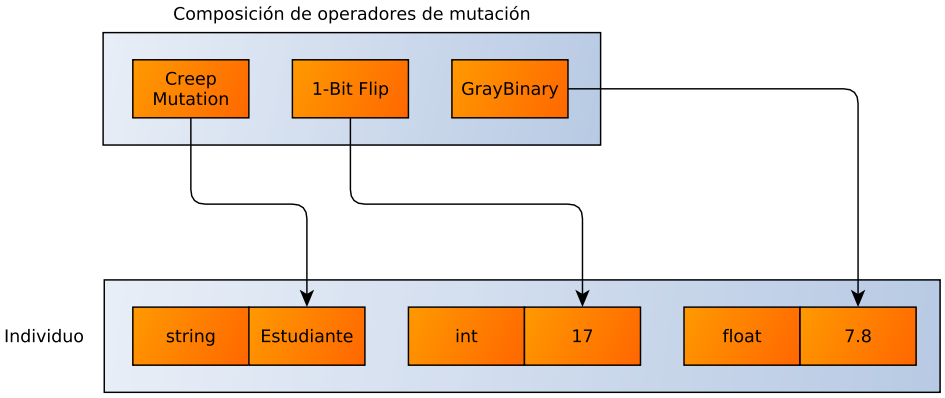


Figura 5.13: Operador de mutación *Composite para elegir mutación por posiciones*.

5.2.7. Operador de mutación *Composite* para elegir mutación por posiciones

Con este operador de mutación podemos determinar qué operadores de mutación se tienen que aplicar para cada posición de cada una las componentes que conforman los individuos de la población.

En la figura 5.13 podemos observar un ejemplo ilustrativo.

De esta manera, gracias a este operador no se exige que se tenga que aplicar el mismo operador de mutación a todas las componentes de los individuos. En lugar de esto, se pueden combinar todos los operadores de mutación que se deseen de acuerdo a la posición que ocupe cada componente dentro del individuo.

5.2.8. Operador de mutación *Random Resetting*

Este operador de mutación reemplaza valores enteros y flotantes por nuevos valores elegidos aleatoriamente entre su máximos y mínimos valores permitidos, de acuerdo con una distribución uniforme.

5.2.9. Operador de mutación *Creep Mutation*

En este operador de mutación dados los valores de las componentes de un individuo y las componentes de dicho individuo que deben ser mutadas, se recorre con un bucle cada una de dichas componentes y se mutan de acuerdo a su tipo.

Cada uno de los distintos tipos posibles se muta de una manera determinada.

- Un valor de tipo *float* se muta de la siguiente manera:
  - Si se ha especificado una lista de posibles valores:
    - Se genera un entero aleatorio p uniformemente distribuido en el intervalo [0, n), siendo n la cantidad de valores posibles disponibles.
    - Se muta el valor original por el valor correspondiente a la posición p de la lista de valores disponibles.
  - En caso de que no se haya especificado una lista de posibles valores:
    - Se muta de acuerdo a la siguiente fórmula:

$$\hat{v} = v + \text{aleatorio-uniforme}(-1, 1) \cdot \frac{r}{p_m} \tag{5.1}$$

      Donde v es el valor original, r es un parámetro que modula el rango del cambio que puede sufrir dicho valor y $\hat{v}$ es el valor resultante. Es decir, v es incrementado o decrementado en un valor que es proporcional a la inversa de p_m de acuerdo al signo del valor aleatorio. Por lo tanto, cuanto menor es la probabilidad de mutación, mayor es el incremento o el decremento cuando tienen lugar. Además, se controla que el nuevo valor no exceda los valores mínimos y máximos que se pueden tomar, en el caso de que hayan sido especificados.

- Un valor de tipo *int* se muta de la misma manera que el tipo *float*, siendo un valor entero el incrementado o decrementado.

- Un valor de tipo *string* se muta sustituyéndolo por otro valor posible generado mediante la estrategia (*IStrategy*) empleada.

- En el caso de las listas la mutación depende de si la lista es de tamaño variable y de si la lista está vacía. Se pueden distinguir cuatro casos:
  - Lista de tamaño fijo y no vacía:
    - Se genera un entero aleatorio p uniformemente distribuido en el intervalo [0, n), siendo n el tamaño de la lista.
    - Se muta el elemento correspondiente a la posición p de la lista, de acuerdo a su tipo.
  - Lista de tamaño fijo y vacía: No se puede mutar.
  - Lista de tamaño variable y no vacía: Se genera un booleano aleatorio uniformemente distribuido y se elige entre mutar un elemento de la lista seleccionado aleatoriamente o mutar la longitud l de la lista a una nueva longitud $\hat{l}$ mediante la inserción o eliminación de elementos de acuerdo con la siguiente fórmula:

$$\hat{l} = l + \left\lceil \text{aleatorio-uniforme}(-1, 1) \cdot \max\left(\frac{(b-a)\cdot(1-p_m)}{2}, 1\right)\right\rceil \tag{5.2}$$

    Donde b es el número máximo de elementos que puede tener la lista y a el mínimo. Cabe aclarar que cuando una lista tiene el mínimo número de elementos solo se le puede añadir elementos y si tiene el máximo número de elementos solo se le puede eliminar elementos.
  - Lista de tamaño variable y vacía: Se añaden elementos a la lista siguiendo la fórmula anterior.

  Tanto cuando se añaden como cuando se eliminan elementos se controla que no se exceda de los límites establecidos por los números máximo y mínimo de elementos permitidos. Además, los nuevos elementos añadidos a la lista son generados mediante la estrategia (*IStrategy*) empleada.

- Un valor de tipo tupla se muta de la siguiente manera:
  - Se genera un entero aleatorio p uniformemente distribuido en el intervalo [0, n), siendo n el tamaño de la tupla.
  - Se muta el elemento correspondiente a la posición p de la tupla, de acuerdo a su tipo.

Capítulo 6

Validación

En este capítulo se detalla la validación del sistema, donde se muestran los distintos tipos de pruebas que se han llevado a cabo para comprobar que el sistema cumple con el comportamiento deseado. También se detallan los experimentos realizados y la visualización de los informes generados.

6.1. Pruebas

La fase de pruebas es una parte fundamental en el desarrollo de software, puesto que permite controlar que el sistema realiza correctamente todas sus funciones tal y como se espera. Las pruebas de software sirven a su vez como información sobre como se debe comportar el sistema ante determinadas situaciones y sobre la calidad del software desarrollado, permitiendo la detección y corrección de errores en el sistema. Así mismo, facilita la verificación de que el sistema continúa comportándose de manera correcta cuando se realizan cambios en el mismo.

6.1.1. Plan de pruebas

6.1.1.1. Alcance

Se han desarrollado las pruebas de software pertinentes a todo el código realizado. Así mismo, se han probado los operadores genéticos realizados, tanto de mutación como de cruce. Además, se han realizado también pruebas para comprobar cualquier modificación del sistema e incremento de funcionalidad en el mismo. Además, se han estudiado los distintos resultados obtenidos por la ejecución del sistema para una determinada composición WS-BPEL 2.0.

6.1.1.2. Tiempo y lugar

Durante el desarrollo del proyecto se han implementado las distintas pruebas que se han incluido en el mismo. A medida que se realizaba cualquier modificación, se añadía alguna funcionalidad o se implementaba un nuevo operador genético se incluían las pruebas necesarias.

A razón de los resultados obtenidos en la ejecución de dichas pruebas, se subsanaban los errores en caso de que presentasen algunos. En caso contrario, las pruebas garantizan que el código implementado cumple con el comportamiento deseado y permite continuar con el desarrollo del resto del proyecto.

Una vez se realizan todas las partes del proyecto necesarias y sus correspondientes pruebas, para comprobar el correcto funcionamiento del sistema se ejecutan todas las pruebas de manera global.

6.1.1.3. Naturaleza de las pruebas

Las pruebas realizadas durante el desarrollo de este proyecto han sido tanto pruebas de caja blanca como pruebas de caja negra.

Las pruebas de caja blanca, también llamadas pruebas estructurales, están dirigidas a comprobar que los detalles de la estructura interna del software a probar están correctamente implementados. Las pruebas de caja blanca están estrechamente unidas a la implementación concreta a probar, por lo que su diseño depende en gran medida del código fuente que va a ser objeto de pruebas. En este sentido, este tipo de pruebas se encuentra intensamente supeditado a los cambios que se produzcan en el código fuente.

Las pruebas de caja negra son pruebas funcionales que se centran en verificar que la salida producida por un determinado módulo es la esperada. Es decir, están dirigidas a comprobar que dados unos datos de entradas, se obtienen unos datos de salida correctos sin importar como se ha comportado internamente el módulo que se esté probando, por lo que se evalúa qué es lo que hace y no cómo lo hace. Este tipo de pruebas es más fácil de mantener ya que no dependen en gran medida la estructura que siga el código fuente, por lo que no le afectan tanto los cambios en el mismo. Estas pruebas facilitan la verificación de que los distintos módulos del programa cumplen con su funcionalidad incluso para valores límites que propician situaciones particulares.

6.1.2. Diseño de las pruebas

Para la realización de las distintas pruebas unitarias se ha empleado el *framework* JUnit. Se trata de un conjunto de bibliotecas empleadas que permite la realización de pruebas unitarias repetibles de aplicaciones Java, de una forma sencilla y controlada.

Las pruebas unitarias permiten verificar que cada uno de los distintos módulos que componen el sistema cumplen correctamente por separado con el comportamiento esperado. Además, facilita la comprobación forma automática en las clases cohesionadas el procedimiento seguido ante situaciones particulares, así como la detección de posibles errores, ya que gracias a las pruebas unitarias éstos se encuentran más acotados y gracias a ellas son localizables de una manera más sencilla. Es conveniente que estas pruebas unitarias abarquen la mayor parte posible del módulo que se esté probando y que sean independientes y automáticas.

Podemos enmarcar todas las pruebas realizadas a los distintos módulos del programa en varios grupos diferenciados:

6.1.2.1. Pruebas de validación

Se trata de pruebas que comprueban que los parámetros de configuración con los que trabajan cada una de las clases desarrolladas en el sistema, se encuentran enmarcados dentro del rango de opciones válidas para cada uno de ellos. Garantizando de esta manera que ninguno de los distintos módulos del sistema se va a comportar de manera extraña por recibir de entrada valores no permitidos para los parámetros con los que trabaja. Además, en el caso de que uno de esos parámetros esté mal definido, se lanzará la correspondiente excepción de *InvalidConfigurationException*.

A continuación podemos observar a modo de ejemplo las pruebas de validación empleadas para el nuevo operador de mutación desarrollado para listas con probabilidad personalizada. En dichas pruebas, se verifica que cada una de las tres probabilidades que se tienen en cuenta en dicho operador para llevar a cabo la mutación, se encuentran dentro del rango de valores permitidos para ellas. En este sentido, gracias a estas pruebas se puede confirmar que:

- La probabilidad de eliminación es mayor o igual que cero.

Listado 6.1: Prueba de validación sobre el límite inferior de la probabilidad de eliminación para el operador de mutación de listas con probabilidad personalizada

```
@Test(expected = InvalidConfigurationException.class)
public void negativeProbabilityElimination() throws Exception {
        mutation.setProbabilityElimination(-0.1);
        mutation.validate();
}
```

- La probabilidad de eliminación es menor o igual que uno.

Listado 6.2: Prueba de validación sobre el límite superior de la probabilidad de eliminación para el operador de mutación de listas con probabilidad personalizada

```
@Test(expected = InvalidConfigurationException.class)
public void greaterThanOneProbabilityElimination() throws Exception {
        mutation.setProbabilityElimination(1.1);
        mutation.validate();
}
```

- La probabilidad de inserción es mayor o igual que cero.

Listado 6.3: Prueba de validación sobre el límite inferior de la probabilidad de inserción para el operador de mutación de listas con probabilidad personalizada

```
@Test(expected = InvalidConfigurationException.class)
public void negativeProbabilityInsertion() throws Exception {
        mutation.setProbabilityInsertion(-0.1);
        mutation.validate();
}
```

- La probabilidad de inserción es menor o igual que uno.

Listado 6.4: Prueba de validación sobre el límite inferior de la probabilidad de inserción para el operador de mutación de listas con probabilidad personalizada

```
@Test(expected = InvalidConfigurationException.class)
public void greaterThanOneProbabilityInsertion() throws Exception {
        mutation.setProbabilityInsertion(1.1);
        mutation.validate();
}
```

- La suma de la probabilidad de eliminación y la probabilidad de inserción es menor o igual que uno.

Listado 6.5: Prueba de validación sobre el límite superior de la suma de las probabilidades de eliminación e inserción para el operador de mutación de listas con probabilidad personalizada

```
@Test(expected = InvalidConfigurationException.class)
public void greaterThanOneSumOfInsertionAndEliminationProbabilities()
    throws Exception {
        mutation.setProbabilityElimination(0.6);
        mutation.setProbabilityInsertion(0.5);
```

```
5           mutation.validate();
6   }
```

- La probabilidad de modificar un elemento es mayor o igual que cero.

Listado 6.6: Prueba de validación sobre el límite inferior de la probabilidad de modificación para el operador de mutación de listas con probabilidad personalizada

```
1   @Test(expected = InvalidConfigurationException.class)
2   public void negativeMutationEliminationProbability() throws Exception
        {
3           mutation.setProbabilityMutationElement(-0.1);
4           mutation.validate();
5   }
```

- La probabilidad de modificar un elemento es menor o igual que uno.

Listado 6.7: Prueba de validación sobre el límite superior de la probabilidad de modificación para el operador de mutación de listas con probabilidad personalizada

```
1   @Test(expected = InvalidConfigurationException.class)
2   public void greaterThanOneMutationEliminationProbability() throws
        Exception {
3           mutation.setProbabilityMutationElement(1.1);
4           mutation.validate();
5   }
```

6.1.2.2. Pruebas funcionales

Con este tipo de pruebas tratamos de asegurar que el funcionamiento del sistema es el adecuado en cada uno de sus módulos, es decir, que produce las salidas esperadas ante unas entradas predeterminadas. Gracias a estas pruebas, podemos comprobar que su funcionamiento es correcto incluso para valores límites, que son los que más probabilidades tienen de causar situaciones extraordinarias en el funcionamiento del sistema.

Cabe recalcar que con este tipo de pruebas no se evalúa el sistema estructuralmente. Es decir, no se evalúa cómo se comporta el sistema internamente, sino qué hace el sistema. En este sentido, las pruebas de este tipo se basan en proporcionar unos datos de entrada y comprobar que los resultados obtenidos coinciden con los esperados.

A continuación podemos observar a modo de ejemplo las pruebas funcionales empleadas para un nuevo operador de mutación desarrollado. En este caso utilizaremos como ejemplo el operador de mutación en código de Gray. En las pruebas desarrolladas para este operador se verifica que dicho operador devuelva un número mutado de forma adecuada y que la conversión de binario natural a codificación de Gray y viceversa se realice correctamente. Mediante las pruebas de este tipo desarrolladas para este operador podemos comprobar que:

- La conversión de binario natural a codificación de Gray se realiza correctamente.

Listado 6.8: Pruebas sobre la conversión de binario natural a codificación de Gray con distintos valores

```
1   @Test
```

```
2  public void testZeroBinaryToGray() {
3          final String binaryString = new String("0");
4          assertDoCorrectBinaryToGray(binaryString, binaryString);
5  }
6
7  @Test
8  public void testOneBinaryToGray() {
9          final String binaryString = new String("1");
10         assertDoCorrectBinaryToGray(binaryString, binaryString);
11 }
12
13 @Test
14 public void testTenThousandBinaryToGray() {
15         final String binaryString = new String("10011100010000"); //
               10000
16         final String expectedGrayString = new String("11010010011000")
               ; //13464
17         assertDoCorrectBinaryToGray(binaryString, expectedGrayString);
18 }
19
20 @Test
21 public void testOneHundredThousandBinaryToGray() {
22         final String binaryString = new String("11000011010100000");
               //100000
23         final String expectedGrayString = new String("
               10100010111110000"); //83440
24         assertDoCorrectBinaryToGray(binaryString, expectedGrayString);
25 }
26
27 private void assertDoCorrectBinaryToGray(final String binaryString,
       final String expectedGrayString){
28         final BigInteger binaryValue = new BigInteger(binaryString,2);
29         final BigInteger grayValue = mutation.binaryToGray(binaryValue
               );
30         assertEquals("The conversion to Gray should be correct",
               expectedGrayString, grayValue.toString(2));
31 }
```

- La conversión de codificación de Gray a binario natural se realiza correctamente.

Listado 6.9: Pruebas sobre la conversión de codificación de Gray a binario natural con distintos valores

```
1  @Test
2  public void testZeroGrayToBinary() {
3          final String grayString = new String("0");
4          assertDoCorrectGrayToBinary(grayString, grayString);
5  }
6
7  @Test
8  public void testOneGrayToBinary() {
9          final String grayString = new String("1");
10         assertDoCorrectGrayToBinary(grayString, grayString);
11 }
12
```

```
13  @Test
14  public void testTenThousandGrayToBinary() {
15          final String grayString = new String("10011100010000"); //
                10000
16          final String expectedBinaryString = new String("11101000011111
                "); //14879
17          assertDoCorrectGrayToBinary(grayString, expectedBinaryString);
18  }
19
20  @Test
21  public void testOneHundredThousandGrayToBinary() {
22          final String grayString = new String("11000011010100000"); //
                100000
23          final String expectedBinaryString = new String("
                10000010011000000"); //66752
24          assertDoCorrectGrayToBinary(grayString, expectedBinaryString);
25  }
26
27  private void assertDoCorrectGrayToBinary(final String grayString,
        final String expectedBinaryString){
28          final BigInteger grayValue = new BigInteger(grayString,2);
29          final BigInteger binaryValue = mutation.grayToBinary(grayValue
                );
30          assertEquals("The conversion to Binary should be correct",
                expectedBinaryString, binaryValue.toString(2));
31  }
```

- La mutación se realiza de correctamente, tanto de números enteros como flotantes.

Listado 6.10: Pruebas sobre la mutación en código de Gray sobre distintos valores enteros y flotantes

```
1   @Test
2   public void testDoMutationIntFirst() {
3           final BigInteger oldValue = new BigInteger("10011100010000",2)
                ;         //10000
4           final BigInteger minValue = new BigInteger("-11000011010100000
                ",2);     //-100000
5           final BigInteger maxValue = new BigInteger("11000011010100000"
                ,2);      //100000
6           final TypeInt type = new TypeInt(minValue, maxValue);
7           final double probability = 2.0;
8           final BigInteger newValue = mutation.doMutationInt(probability
                , type, oldValue);
9           assertTrue("The mutation should have produced a new object
                with a different value", !oldValue.equals(newValue));
10          assertTrue("The mutated value should respect minimum value
                constraints", newValue.compareTo(minValue) >= 0);
11          assertTrue("The mutated value should respect maximum value
                constraints", newValue.compareTo(maxValue) <= 0);
12  }
13
14  @Test
15  public void testDoMutationIntSecond() {
16          final BigInteger oldValue = new BigInteger("10000010000",2);
```

```
            //1040
17          final BigInteger minValue = new BigInteger("1111111111",2); //
               1023
18          final BigInteger maxValue = new BigInteger("101111111110",2);
               //3070
19          final TypeInt type = new TypeInt(minValue, maxValue);
20          final double probability = 2.0;
21          final BigInteger newValue = mutation.doMutationInt(probability
               , type, oldValue);
22          assertTrue("The mutation should have produced a new object
               with a different value", !oldValue.equals(newValue));
23          assertTrue("The mutated value should respect minimum value
               constraints", newValue.compareTo(minValue) >= 0);
24          assertTrue("The mutated value should respect maximum value
               constraints", newValue.compareTo(maxValue) <= 0);
25  }
26
27  @Test
28  public void testDoMutationFloatFirst() {
29          final BigDecimal oldValue = new BigDecimal(new BigInteger("
               10000"),4);
30          final BigDecimal minValue = new BigDecimal(new BigInteger("
               -100000"),5);
31          final BigDecimal maxValue = new BigDecimal(new BigInteger("
               100000"),5);
32          final TypeFloat type = new TypeFloat(minValue, maxValue);
33          final double probability = 1.0;
34          final BigDecimal newValue = mutation.doMutationFloat(
               probability, type, oldValue);
35          assertTrue("The mutation should have produced a new object", !
               oldValue.equals(newValue));
36          assertTrue("The mutated value should respect minimum value
               constraints", newValue.compareTo(minValue) >= 0);
37          assertTrue("The mutated value should respect maximum value
               constraints", newValue.compareTo(maxValue) <= 0);
38  }
39
40  @Test
41  public void testDoMutationFloatSecond() {
42          final BigDecimal oldValue = new BigDecimal(new BigInteger("104
               "),2);
43          final BigDecimal minValue = new BigDecimal(new BigInteger("
               1023"),3);
44          final BigDecimal maxValue = new BigDecimal(new BigInteger("307
               "),2);
45          final TypeFloat type = new TypeFloat(minValue, maxValue);
46          final double probability = 1.0;
47          final BigDecimal newValue = mutation.doMutationFloat(
               probability, type, oldValue);
48          assertTrue("The mutation should have produced a new object", !
               oldValue.equals(newValue));
49          assertTrue("The mutated value should respect minimum value
               constraints", newValue.compareTo(minValue) >= 0);
50          assertTrue("The mutated value should respect maximum value
               constraints", newValue.compareTo(maxValue) <= 0);
51  }
```

```
52
53  @Test
54  public void testTwoConversions() {
55          final String binaryString = new String("10011100010000"); //
                10000
56          final BigInteger oldValue = new BigInteger(binaryString,2);
57          final BigInteger grayValue = mutation.binaryToGray(oldValue);
58          final BigInteger newValue = mutation.grayToBinary(grayValue);
59          assertEquals("A number converted to gray and then converted to
                 binary should be the same.", binaryString, newValue.
                toString(2));
60  }
```

6.1.2.3. Pruebas estructurales

Gracias a este tipo de pruebas podemos comprobar los detalles de la estructura interna de los distintos módulos del sistema. En este sentido, facilita la verificación de que el software a probar está bien implementado.

Estas pruebas son especialmente útiles para probar código en el que se generan elementos aleatoriamente y se toman decisiones a partir de ellos, ya que con pruebas convencionales no es posible tener bajo control todo el abanico de opciones posibles. Es por ello que en este sistema este tipo de pruebas son bastante útiles, puesto que es habitual generar números aleatorios para seleccionar posiciones o para comparar con probabilidades de aplicación de determinados operadores genéticos, así como también es habitual generar booleanos aleatorios para tomar decisiones simulando el lanzamiento de una moneda.

Para la realización de este tipo de pruebas se ha empleado *mockito* [19]. Se trata de un *framework* de pruebas para código Java, que permite la creación de objetos *mocks* para la prueba software. Es decir, permite la utilización de objetos que simulan el comportamiento de una clase para la realización de pruebas y posibilita realizar comprobaciones como cuantas veces ha sido llamado un método determinado.

A continuación se muestra a modo de ejemplo las pruebas estructurales empleadas para el nuevo operador de cruce de doble punto. En las pruebas desarrolladas para este operador se verifica la manera en la que el operador garantiza que los dos puntos de cruce empleados por el operador sean puntos de cruce válidos, repitiendo la generación aleatoria de los puntos de cruce tantas veces como sea necesario.

A través de las pruebas de este tipo desarrolladas para este operador podemos comprobar que:

- Se reintenta la generación aleatoria de los puntos de cruce cuando los generados coinciden, puesto que no se realizaría intercambio de componentes entre los dos individuos implicados en el cruce.

Listado 6.11: Pruebas sobre la mutación en código de Gray sobre distintos valores enteros y flotantes

```
1  @Test
2  public void applyRetriesForSamePoints() throws Exception {
3          final Random r = mock(Random.class);
4          when(r.nextInt(5)).thenReturn(2).thenReturn(2).thenReturn(1).
               thenReturn(2);
5          crossover.setPRNG(r);
6
7          crossover.apply(new GAGeneticOperatorOptions(), father, mother
               );
8          verify(r, times(4)).nextInt(5);
9
10         assertCrossoverIsCorrect(1, 2, father, mother, left, right);
```

```
11  }
12
13  private static void assertCrossoverIsCorrect(final int position1,
        final int position2,
14                          GgenIndividual father, GgenIndividual mother,
15                          GgenIndividual leftChild, GgenIndividual
                                rightChild) {
16          int cont = 0;
17          while (cont < position1) {
18                  assertEquals(father.getComponent(cont), leftChild.
                        getComponent(cont));
19                  assertEquals(mother.getComponent(cont), rightChild.
                        getComponent(cont));
20                  cont++;
21          }
22          while (cont < position2) {
23                  assertEquals(mother.getComponent(cont), leftChild.
                        getComponent(cont));
24                  assertEquals(father.getComponent(cont), rightChild.
                        getComponent(cont));
25                  cont++;
26          }
27          while (cont < father.getNumberComponents()) {
28                  assertEquals(father.getComponent(cont), leftChild.
                        getComponent(cont));
29                  assertEquals(mother.getComponent(cont), rightChild.
                        getComponent(cont));
30                  cont++;
31          }
32  }
```

- Se reintenta la generación aleatoria de los puntos de cruce cuando los generados son las dos posiciones extremas del individuo, puesto que implicaría un intercambio de todos los componentes entre los dos individuos implicados en el cruce y daría lugar a dos individuos idénticos a los padres.

Listado 6.12: Pruebas sobre la mutación en código de Gray sobre distintos valores enteros y flotantes

```
1   @Test
2   public void applyRetriesForPointsInLimits() throws Exception {
3           final Random r = mock(Random.class);
4           when(r.nextInt(5)).thenReturn(0).thenReturn(4).thenReturn(1).
                thenReturn(2);
5           crossover.setPRNG(r);
6
7           crossover.apply(new GAGeneticOperatorOptions(), father, mother
                );
8           verify(r, times(4)).nextInt(5);
9
10          assertCrossoverIsCorrect(1, 2, father, mother, left, right);
11  }
```

6.1.2.4. Pruebas de integración y aceptación

Con las pruebas de integración se prueba la interacción entre los módulos que componen el sistema, con el objetivo de verificar que funcionan correctamente en conjunto. Se realizan una vez se han comprobado mediante pruebas unitarias el funcionamiento de todos los módulos por separado y sirven para garantizar que funcionan juntos dentro del marco del sistema.

Se realizan ejecuciones completas para comprobar el correcto funcionamiento de todo el sistema, evaluando que no se haya producido ningún error, así como los resultados obtenidos.

Estas pruebas también han servido como pruebas de aceptación, el objetivo de las pruebas de aceptación es comprobar que el sistema cumple con la funcionalidad requerida por el cliente, en este caso los investigadores del grupo de investigación UCASE. Es decir, sirven para validar que el sistema se comporta tal y como esperan los investigadores para poder emplearlo en estudios de investigación.

Haciendo uso de este tipo de experimentos se ha realizado un artículo que ha sido presentado y aceptado a las XX Jornadas de Ingeniería del Software y Bases de Datos. En la sección Experimentos 6.2 se trata este tema con más detalle.

6.2. Experimentos

Se han realizado experimentos sobre composiciones WS-BPEL 2.0, de manera que su estudio ha servido para validar y mejorar la implementación del algoritmo genético diseñado. De manera que el sistema esté bien definido y sea de utilidad para alcanzar resultados de investigación significativos.

6.2.1. Artículo de investigación

Se ha presentado un artículo de investigación a las XX Jornadas de Ingeniería del Software y Bases de Datos, llamado «Análisis y determinación del impacto del operador de mutación en la generación genética de casos de prueba para WS-BPEL», el cual ha sido aceptado para su publicación. En este artículo se analizan los resultados obtenidos por el algoritmo genético mediante la aplicación de distintos operadores de mutación sobre la composición WS-BPEL 2.0 llamada *Triangle*.

Para llevar a cabo dicho artículo se han estudiado los resultados obtenidos por tres de los operadores de mutación de los que dispone el sistema: *Creep Mutation*, *Random Resetting* y *1-Bit Flipping*, que se aplican a la composición *Triangle*. Además se ha limitado el tamaño de los lados del triángulo a tres rangos diferentes: 0-127, 0-255 y 0-2047, para investigar con qué rango de valores para los lados del triángulo se obtienen mejores resultados.

Los resultados obtenidos para los distintos operadores de mutación en los experimentos realizados para dicho artículos de investigación se pueden observar en las tablas 6.1, 6.2 y 6.5.

Tabla 6.1: Tabla de puntuación de mutación para el conjunto de casos de prueba inicial aleatorio y los conjuntos generados por el algoritmo genético. En las columnas: CM (Creep Mutation), RR (Random Resetting), 1-BF (1-Bit Flipping).

	Aleat.(%)	AG (%)			AG (siembra) (%)		
Rango		CM	RR	1-BF	CM	RR	1-BF
0-127	80.3	89.8	90.9	91.3	96.6	96.2	96.8
0-255	75.9	81.4	76.1	79.5	95.8	95.6	96.6
0-2047	70.1	70.6	70.5	70.1	96.2	95.6	95.8

Como se puede observar, además se comparan los resultados obtenidos para un conjunto de casos de prueba inicial sin siembra y con siembra automática. En el primer caso, el conjunto de caso de prueba que el algoritmo genético emplea como población inicial es generado aleatoriamente, mientras que en

Tabla 6.2: Tabla de cobertura de sentencias para el conjunto de casos de prueba inicial aleatorio y los conjuntos generados por el algoritmo genético. En las columnas: CM (Creep Mutation), RR (Random Resetting), 1-BF (1-Bit Flipping).

	Aleat.(%)	AG (%)			AG (siembra) (%)		
Rango		CM	RR	1-BF	CM	RR	1-BF
0-127	82.6	91.3	91.3	91.3	100.0	100.0	100.0
0-255	82.6	84.8	82.6	82.6	100.0	100.0	100.0
0-2047	73.9	76.1	73.9	73.9	100.0	100.0	100.0

Tabla 6.3: Tabla de cobertura de condición/decisión para el conjunto de casos de prueba inicial aleatorio y los conjuntos generados por el algoritmo genético. En las columnas: CM (Creep Mutation), RR (Random Resetting), 1-BF (1-Bit Flipping).

	Aleat.(%)	AG (%)			AG (siembra) (%)		
Rango		CM	RR	1-BF	CM	RR	1-BF
0-127	61.5	76.9	80.8	84.6	92.3	90.4	90.4
0-255	61.5	69.2	61.5	69.2	92.3	92.3	92.3
0-2047	53.8	53.8	53.8	53.8	88.5	88.5	92.3

el caso de utilizar siembra se cambian una serie de casos de pruebas del conjunto aleatorio por casos de prueba especialmente diseñados para el problema a estudiar.

Por otro lado, los experimentos se han realizado estableciendo una probabilidad de mutación p_m al 10 % y una probabilidad de cruce p_c al 90 %.

6.2.2. Experimentos adicionales

Además, se han realizado más experimentos, con el objetivo de probar y mejorar la herramienta y los operadores empleados.

Por ejemplo, se han realizado experimentos empleando el operador de mutación específico para la composición *Triangle* con el objetivo de comparar los resultados del resto de operadores con los resultados de un operador de mutación desarrollado específicamente para la mutación de triángulos.

Podemos ver los resultados en la siguiente tabla:

Tabla 6.4: Tabla de puntuación de mutación para los conjuntos de casos de prueba generados por el algoritmo genético con el operador TMO (*TriangleMutationOperator*).

	AG (%)
Rango	TMO
0-127	98.86

También se han realizado experimentos empleando la composición *BMI*, aplicando el algoritmo genético con el operador de mutación *1-Bit Flip*. En la siguiente tabla podemos observar los resultados:

6.2.3. Generación del conjunto de casos de prueba inicial

El conjunto de casos de prueba que conforma la población inicial empleada por el algoritmo genético es generada aleatoriamente. De esta manera, con el objetivo de intentar disminuir en la medida de lo posible que los resultados puedan variar de una ejecución a otra con distintas poblaciones iniciales, se sigue el siguiente procedimiento para seleccionar la población inicial de entre todas las posibles:

Tabla 6.5: Tabla de puntuación de mutación para los conjuntos de casos de prueba generados por el algoritmo genético con el operador *1-Bit Flip* a la composición *BMI*.

	AG (%)
Composción	*1-Bit Flip*
BMI	96.67

- Se define el tamaño que debe tener la población. Para ello se toma entre un 20 % y un 25 % del número total de mutantes generados a partir de la composición WS-BPEL, con un mínimo de 10 y un máximo de 50.
- Se generan 30 conjuntos de casos de prueba aleatorios en relación al tamaño de la población definido.
- Se ejecutan los 30 conjuntos de casos de prueba aleatorios contra la composición original y los mutantes generados a partir de ella para medir las puntuaciones de mutación y las coberturas de sentencias y de decisión/condición.
- Para cada uno de los 30 conjuntos, se calculan las medianas para las métricas de calidad anteriores.
- Se selecciona un conjunto de casos de prueba que se sitúe en la mediana de los resultados obtenidos.

Gracias a este procedimiento conseguimos un buen candidato para la población inicial empleada por el algoritmo genético, puesto que conseguimos un conjunto de casos de prueba generado aleatoriamente cerciorándonos de que no está sesgado.

6.2.4. Composición estudiadas

Al grupo de investigación UCASE le han sido facilitadas una serie de composiciones por el grupo de investigación del Prof. Franz Wotawa de la Technische Universität Graz. Se ha realizado la adaptación de estas composiciones para que puedan ser empleadas correctamente por el algoritmo genético para los diversos experimentos realizados.

En concreto, se han realizado estudios con las siguiente composiciones:

6.2.4.1. Composición *Triangle*

Esta composición consta de 297 líneas de código y genera 298 mutantes, siendo 34 de ellos equivalentes.

El objetivo de la composición *Triangle* es la clasificación de triángulos de acuerdo a la longitud de cada uno de sus lados. En este sentido, la clasificación que se realiza en esta composición distingue distintos tipos de triángulos según las propiedades mostradas a continuación:

- La longitud de alguno de los lados no es mayor que 0. En este caso sería un triángulo no válido.
- La longitud alguno de los lados es mayor que la suma de los otros dos. En este caso también se trataría de un triángulo no válido.
- Los tres lados tienen la misma longitud.
- Dos de los lados tienen la misma longitud.
- Todos los lados tienen una longitud distinta.

En la figura 6.1 podemos observar el comportamiento que sigue la composición *Triangle*.

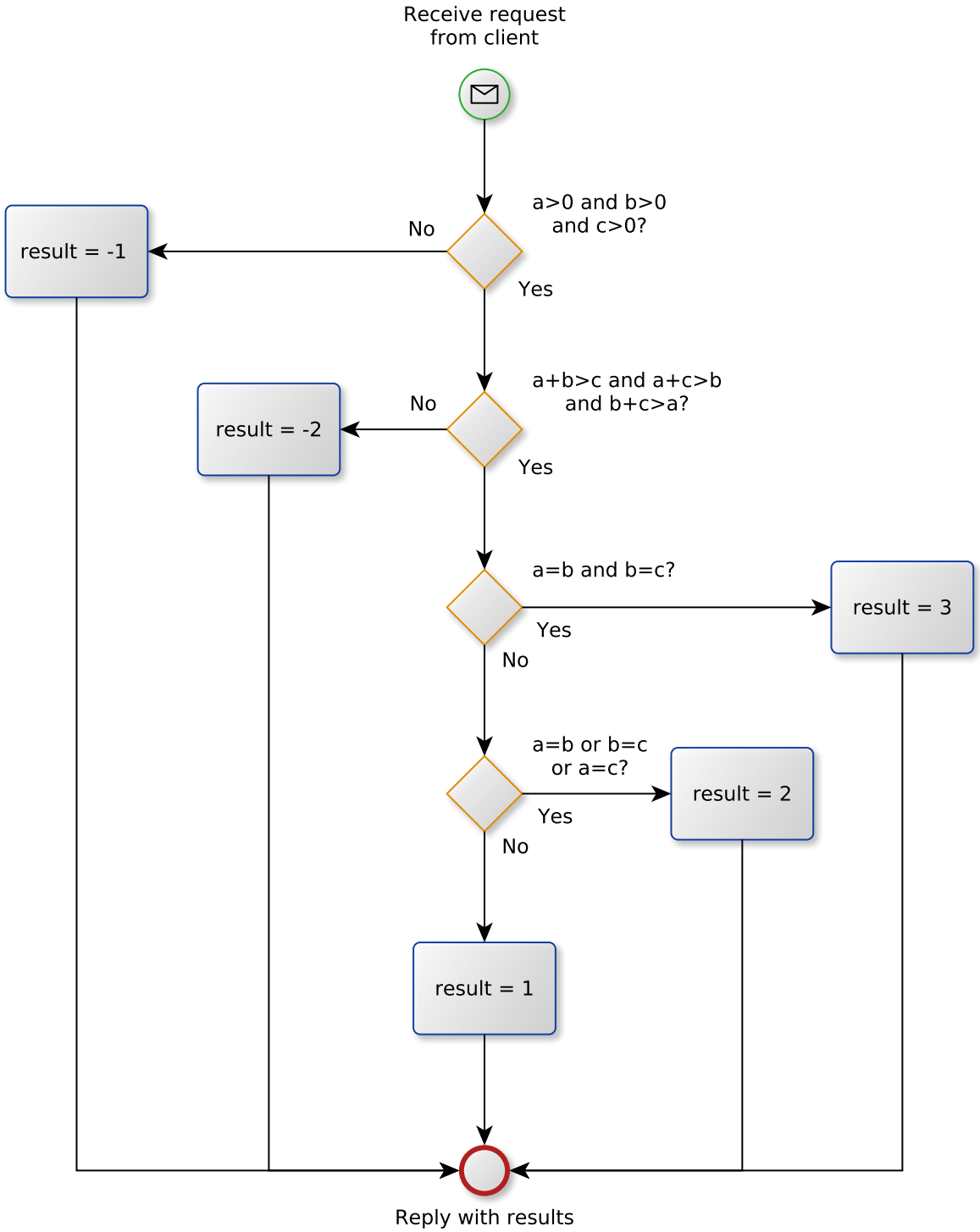


Figura 6.1: Diagrama de flujo de la composición *Triangle*.

6.2.4.2. Composición *BMI*

La composición *BMI* consta de 230 líneas de código y genera 94 mutantes, siendo 4 de ellos equivalentes.

Esta composición calcula el Índice de Masa Corporal (*BMI*) a través del peso y la altura que recibe como entrada. Puede producir las siguientes salidas:

- Con un *BMI* menor que 18 el resultado devuelto es *underweight*.
- Con un *BMI* menor que 25 el resultado devuelto es *healthy*.
- Con un *BMI* menor que 30 el resultado devuelto es *overweight*.
- Con un *BMI* menor que 40 el resultado devuelto es *obese*.
- Con un *BMI* mayor o igual que 40 el resultado devuelto es *very obese*.

En la figura 6.2 podemos observar el comportamiento que sigue la composición *Bmi*.

6.2.5. Máquinas empleadas para la ejecución de los experimentos

Los experimentos se han realizado con las máquinas que emplea el grupo de investigación UCASE. Se emplean máquinas virtuales VMWare, las características de cada una de estas máquinas son las siguientes:

- CPU: Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz
  - Modelo: Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz
  - Tamaño de caché: 20MB
  - Número de núcleos: 1
- Memoria RAM: 16GB
- Disco duro: 64GB
- Distribución: Ubuntu 14.04.2 LTS
- Kernel: 3.13.0-55-generic

Los experimentos realizados son muy complejos y requieren gran cantidad de tiempo, en concreto, cada una de las ejecuciones de los experimentos realizados sobre la composición *Triangle* para el desarrollo del artículo publicado en las XX Jornadas de Ingeniería del Software y Bases de Datos tardan unas 12 horas de media sin el empleo de semillas, y unas 9 horas de media con el empleo de semillas.

Teniendo en cuenta que cada uno de estos experimentos se compone de 30 ejecuciones. Empleando una sola máquina, cada experimento sin semilla tardaría aproximadamente 15 días y cada experimento con semilla tardaría aproximadamente 11 días.

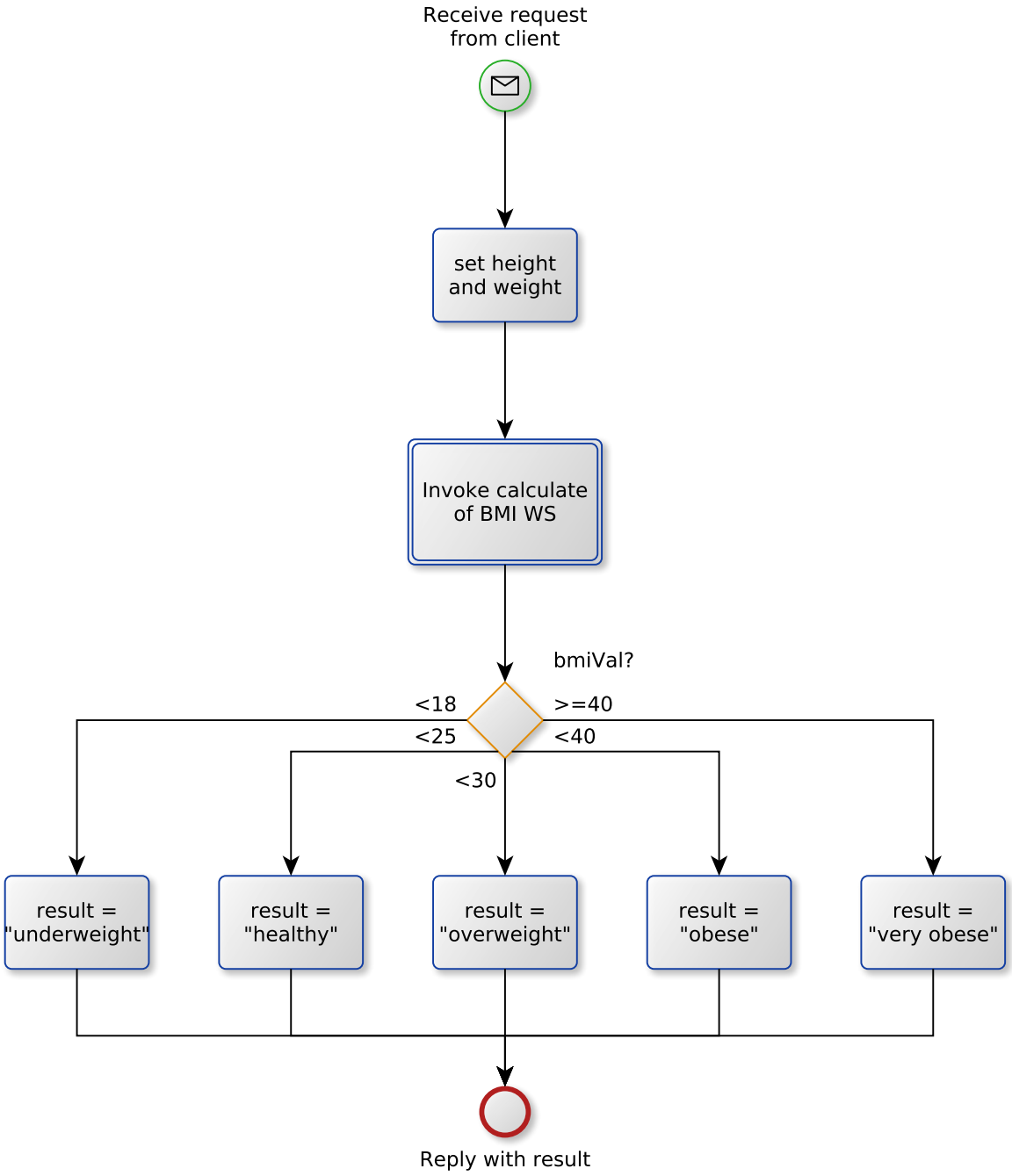


Figura 6.2: Diagrama de flujo de la composición *BMI*.

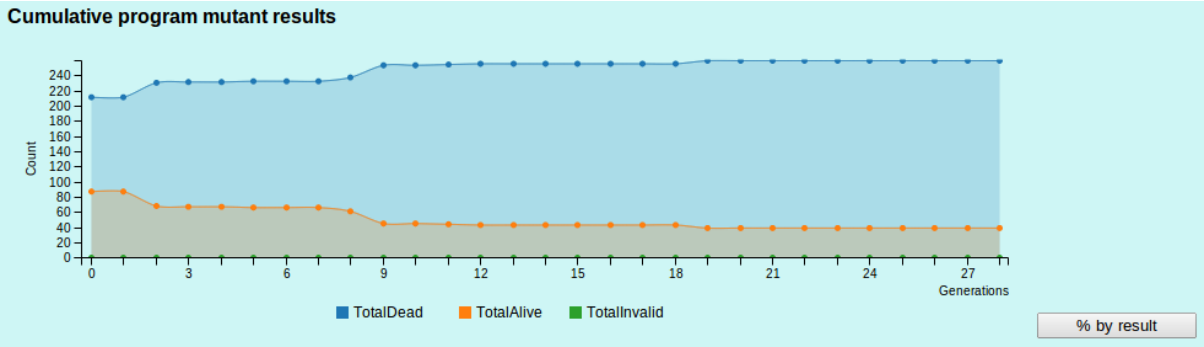


Figura 6.3: Gráfica de acumulación de resultados de mutantes.

6.3. Visualización

Uno de los problemas que radicaban en este sistema es la dificultad que presenta el análisis de los resultados obtenidos por el algoritmo genético, ya que es necesario interpretar manualmente todos los datos, almacenados por los *loggers*, que se han ido generando durante la ejecución del algoritmo. Por lo tanto, analizar los resultados obtenidos por el algoritmo genético resulta una tarea tediosa, en la que el usuario tiene que emplear un tiempo considerable para sacar una conclusión sobre ellos.

Es por ello que se hace de vital importancia disponer de un informe con el que poder observar rápidamente los resultados obtenidos. En este sentido, se ha mejorado la visualización haciendo uso de HTML 5 y JavaScript, realizándose un *logger* que almacena la información necesaria y genera un fichero con extensión *html* en el que se puede observar de una manera clara y rápida el comportamiento que ha tenido el algoritmo durante una ejecución.

Para la generación del fichero *html* se emplea una plantilla Velocity con la que se representa la organización que debe seguir la información, que ha sido almacenada por el *logger*, de acuerdo a la estructura que se ha especificado en la plantilla. Es decir, gracias a la plantilla Velocity se produce código HTML cuyo contenido es la información que ha sido captada por el *logger* durante la ejecución del algoritmo genético.

La información contenida en el fichero *html* se encuentra dividida en bloques. Existe un primer bloque que contiene información general sobre toda la ejecución, en dicho bloque podemos observar:

- Acumulación de mutantes distintos del programa original que han sido matados a lo largo de las generaciones de la ejecución. En esta gráfica podemos observar para cada generación el número de mutantes distintos muertos hasta dicha generación, es decir, muestra la acumulación de los mutantes distintos que han muerto en esa generación y en las anteriores. Por ejemplo, si en una generación solo se ha matado un mutante que no se había matado antes, el valor que se tome en la gráfica para esa generación será el de la generación anterior más uno. Además del número de mutantes muertos también se indica el número de mutantes vivos e inválidos.

  Es una gráfica especialmente útil si se quiere observar, por ejemplo, a partir de que generación se consiguió obtener el número total de mutantes distintos que se han matado en la ejecución. En la figura 6.3 podemos ver un ejemplo de esta gráfica.

  En esta gráfica se dispone de un botón con el que se puede elegir si se quiere observar los datos en porcentajes o la cantidad total.

- Resultados de los mutantes del programa original en cada generación. En esta gráfica podemos ver exactamente el número de mutantes muertos, vivos e inválidos para cada generación de una determinada ejecución. Podemos observar un ejemplo de esta gráfica en la figura 6.4.

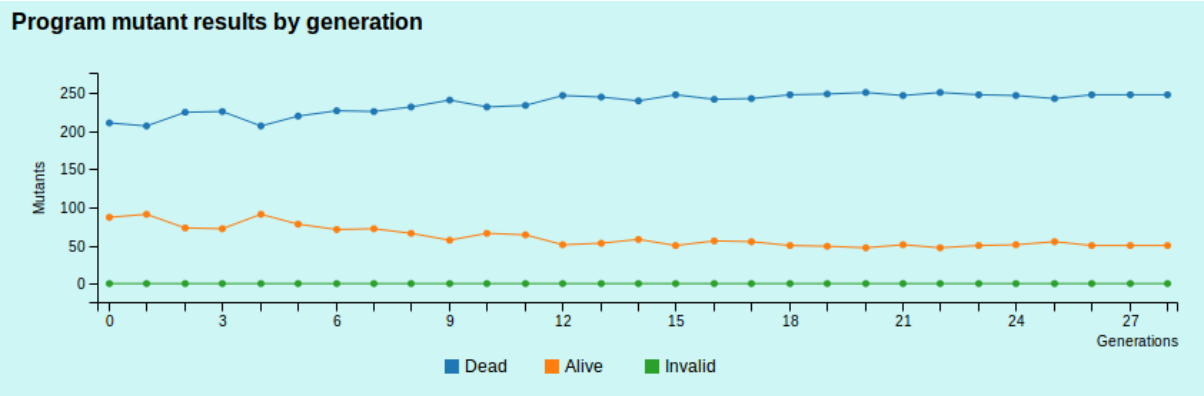


Figura 6.4: Gráfica de resultados por generación.

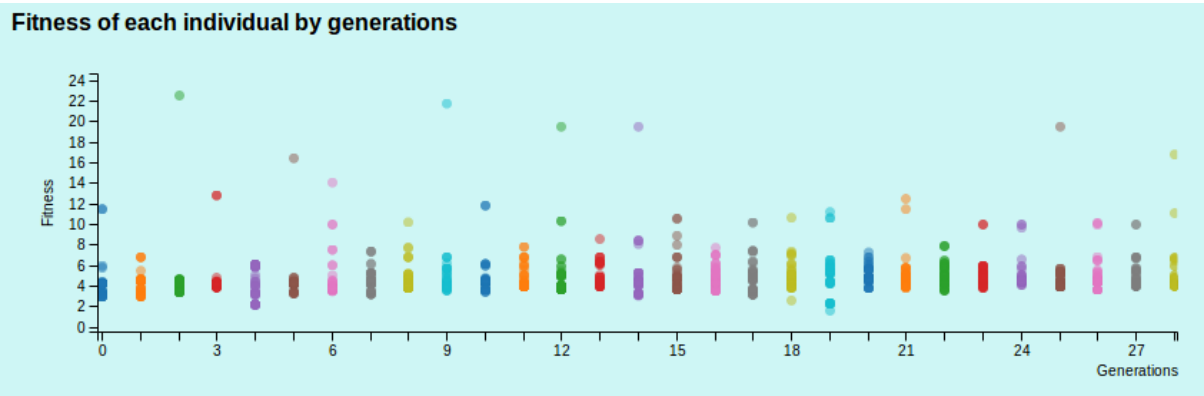


Figura 6.5: Gráfica de aptitud de cada individuo de cada generación.

- Aptitud de los individuos de cada generación. Se trata de una gráfica de puntos en la que podemos observar el valor con respecto a la función de aptitud que toma cada uno de los tests que componen cada generación de la ejecución. Podemos encontrar un ejemplo de este tipo de gráfica en la figura 6.5.
- Condición de terminación de la ejecución: Se muestra la causa por la que la ejecución del algoritmo genético ha terminado, lo que puede ocurrir por varias razones distintas:
  - Estancamiento de la aptitud media.
  - Estancamiento de la aptitud máxima.
  - Ejecución del número máximo de generaciones especificado.
  - Se han conseguido matar todos los mutantes generados a partir del programa original.
  - Se ha conseguido matar un determinado porcentaje de todos los mutantes generados a partir del programa original.

  En la figura 6.6 podemos observar un ejemplo.

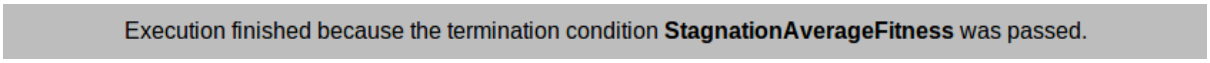


Figura 6.6: Condición de parada cumplida en la ejecución.

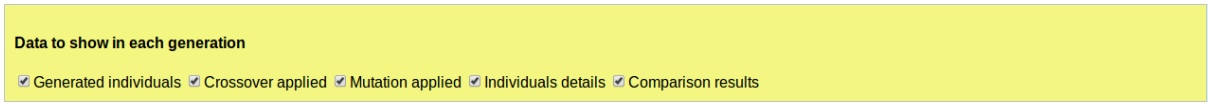


Figura 6.7: Grupo de opciones para cada generación.

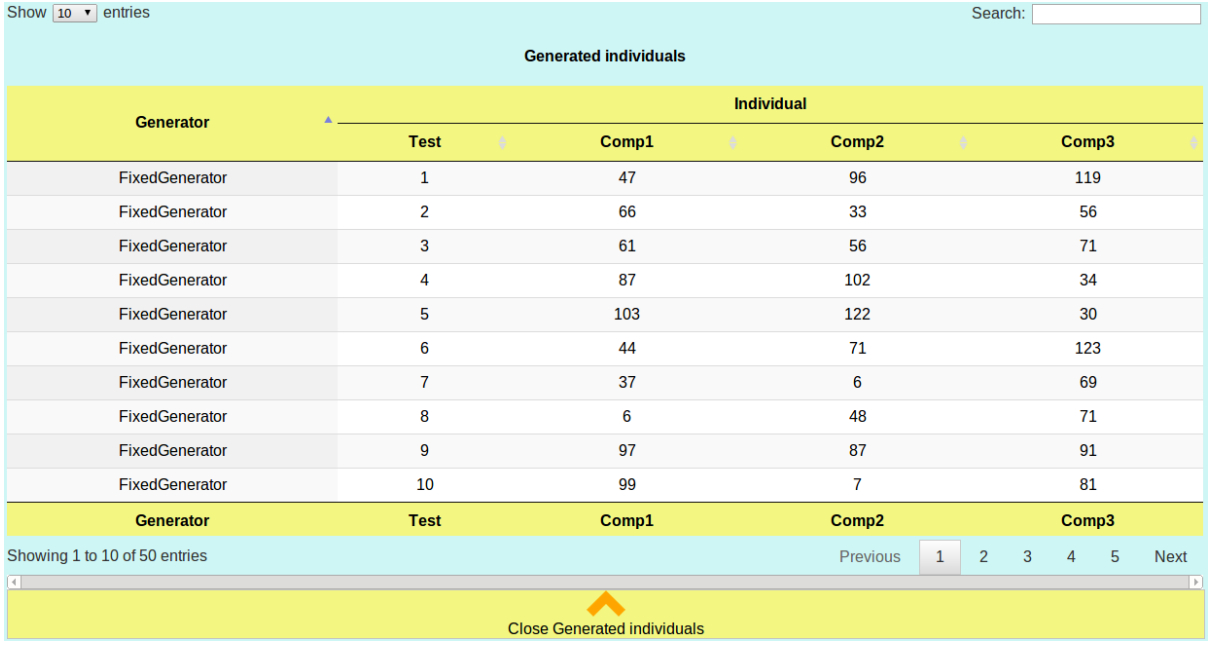


Figura 6.8: Nuevos mutantes generados en una generación.

El resto de bloques en los que se encuentra dividido el informe generado en HTML se corresponden con cada una de las generaciones que han tenido lugar en una determinada ejecución.

Justo antes del conjunto de bloques correspondientes a cada generación, tenemos un grupo de *check-boxes* (véase la figura 6.7) con los que podemos elegir la información que queremos observar en los bloques correspondientes a las distintas generaciones.

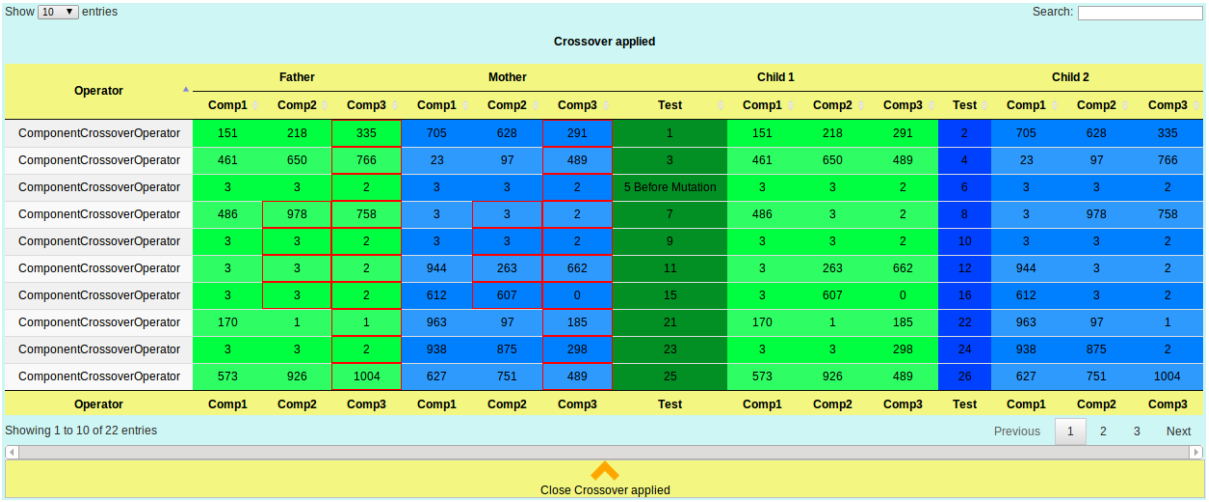
En dicha información podemos encontrar:

- Nuevos mutantes generados. Se trata de una tabla en la que se muestran todos los nuevos mutantes producidos en una determinada generación a través de un generador. Para cada nuevo caso de prueba creado se muestra el generador empleado, el test al que representa dicho caso de prueba y los distintos componentes que lo forman. En la figura 6.8 podemos ver un ejemplo de este tipo de tabla.

- Cruces aplicados. En esta tabla podemos observar todos los cruces que han tenido lugar en una determinada generación. Para cada cruce realizado se muestra el operador de cruce empleado, los componentes del padre, de la madre y de los hijos resultantes, así como el test al que corresponde cada hijo. Podemos ver un ejemplo de este tipo de tabla en la figura 6.9.

  Además, se puede identificar fácilmente los componentes que han participado en el proceso de cruce, puesto que dichos componentes se destacan mediante un borde rojo. De esta manera, se facilita al usuario la distinción de los componentes que se han intercambiado.

  Por otro lado, esta tabla también muestra si los mismos individuos que han sido objeto de cruce, también han sufrido mutación posteriormente o incluso si uno de ellos ha sido descartado por no ser necesario para completar la población, al haberse alcanzado ya el tamaño de la población requerido.



Show 10 entries Search:

Crossover applied

Operator	Father			Mother				Child 1				Child 2		
	Comp1	Comp2	Comp3	Comp1	Comp2	Comp3	Test	Comp1	Comp2	Comp3	Test	Comp1	Comp2	Comp3
ComponentCrossoverOperator	151	218	335	705	628	291	1	151	218	291	2	705	628	335
ComponentCrossoverOperator	461	650	766	23	97	489	3	461	650	489	4	23	97	766
ComponentCrossoverOperator	3	3	2	3	3	2	5 Before Mutation	3	3	2	6	3	3	2
ComponentCrossoverOperator	486	978	758	3	3	2	7	486	3	2	8	3	978	758
ComponentCrossoverOperator	3	3	2	3	3	2	9	3	3	2	10	3	3	2
ComponentCrossoverOperator	3	3	2	944	263	662	11	3	263	662	12	944	3	2
ComponentCrossoverOperator	3	3	2	612	607	0	15	3	607	0	16	612	3	2
ComponentCrossoverOperator	170	1	1	963	97	185	21	170	1	185	22	963	97	1
ComponentCrossoverOperator	3	3	2	938	875	298	23	3	3	298	24	938	875	2
ComponentCrossoverOperator	573	926	1004	627	751	489	25	573	926	489	26	627	751	1004
Operator	Comp1	Comp2	Comp3	Comp1	Comp2	Comp3	Test	Comp1	Comp2	Comp3	Test	Comp1	Comp2	Comp3

Showing 1 to 10 of 22 entries Previous 1 2 3 Next

Close Crossover applied

Figura 6.9: Cruces aplicados en una generación.

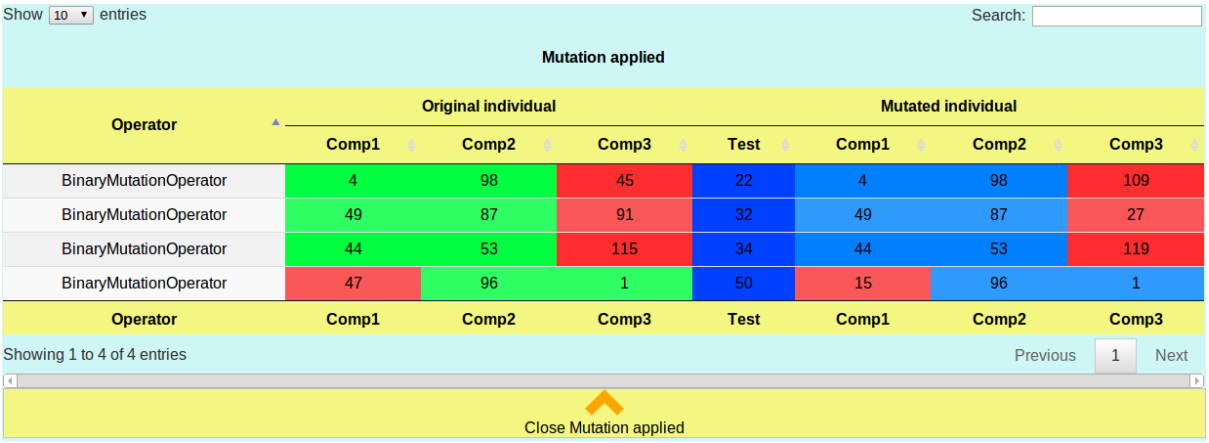
Figura 6.10: Detalles de cruces aplicados en una generación.

En la figura 6.10 podemos ver en detalle estas características comentadas.

- Mutaciones aplicadas. Es una tabla en la que podemos observar todas las mutaciones que se han realizado en una determinada generación. Para cada mutación podemos observar el operador de mutación que se ha utilizado, el test a que corresponde el caso de prueba resultante y los componentes tanto del padre y como del hijo resultante. Además, los componentes en los que han tenido lugar las mutaciones se diferencian muy fácilmente ya que se muestran de un color diferente al resto de componentes. En la figura 6.11 tenemos un ejemplo de esta tabla.

  Al igual que ocurre en el cruce, podemos observar si algún individuo ha sido descartado por haberse completado ya el tamaño de la población requerido. En la figura 6.12 podemos observar en detalle un ejemplo que ilustra esta particularidad.

- Población de la generación: Tabla que muestra las características de cada caso de prueba que



Show 10 entries Search:

Mutation applied

Operator	Original individual				Mutated individual		
	Comp1	Comp2	Comp3	Test	Comp1	Comp2	Comp3
BinaryMutationOperator	4	98	45	22	4	98	109
BinaryMutationOperator	49	87	91	32	49	87	27
BinaryMutationOperator	44	53	115	34	44	53	119
BinaryMutationOperator	47	96	1	50	15	96	1
Operator	Comp1	Comp2	Comp3	Test	Comp1	Comp2	Comp3

Showing 1 to 4 of 4 entries Previous 1 Next

Close Mutation applied

Figura 6.11: Mutaciones aplicadas en una generación.

Figura 6.12: Individuo mutado descartado.

Show 10 entries Search:

Individuals of generation 1

Test	Fitness	Individual		
		Comp1	Comp2	Comp3
1	4.600026315246797	87	102	61
2	4.520480860701342	97	74	34
3	3.686243568626526	109	8	119
4	3.6544815298522924	47	96	48
5	3.1862435686265256	60	26	114
6	2.9776338304115235	126	12	69
7	4.4704808607013415	71	22	85
8	3.1862435686265256	20	33	124
9	4.520480860701342	61	56	25
10	3.386243568626526	8	22	71
Test	Fitness	Comp1	Comp2	Comp3

Showing 1 to 10 of 50 entries Previous 1 2 3 4 5 Next

Close Individuals of generation 1

Figura 6.13: Población de una generación.

conforma la población de una determinada generación. En dicha tabla podemos observar para cada caso de prueba el test al que corresponde, la aptitud del individuo y cada uno de los componentes que lo conforman. Podemos observar un ejemplo de esta tabla en la figura 6.13.

- Matriz de ejecución: En esta tabla nos encontramos la matriz de ejecución de los mutantes del programa original contra cada uno de los tests resultantes de una determinada generación. Es decir, para cada mutante del programa original podemos ver el resultado que ha obtenido con cada uno de los tests que conforman una generación específica.

  Cada mutante del programa original se representa mediante el nombre del operador, el lugar de aplicación y el atributo. Además, se muestra el resultado obtenido para cada uno de los tests, que puede ser: 0 si ha quedado vivo, 1 si ha muerto y 2 si es un mutante inválido. En la figura 6.14 podemos observar un ejemplo de esta tabla.

Cada uno de los bloques en los que se divide el fichero *html* son bloques despegables, de manera que permite una fácil navegación a través del documento. Así mismo, cada una de las tablas presentes en los bloques de las distintas generaciones disponen de una pestaña que permite tanto ocultarlas como mostrarlas.

Para la creación de las tablas se ha empleado *dataTables* [20], que es una herramienta que mejora el acceso a los datos en tablas HTML 5, permitiendo una cómoda ordenación, búsqueda, filtrado y paginación de los datos contenidos en la tabla.

Además, para la creación de las distintas gráficas se ha empleado C3 [21], que facilita la creación de gráficas basadas en D3 [22] permitiendo una fácil personalización del estilo de las gráficas y un sencillo control sobre las mismas debido a la amplia API que proporciona.

Show 10 entries Search:

Mutants	Test1	Test2	Test3	Test4	Test5	Test6	Test7	Test8	Test9	Test10	Test11	Test12	Test13	Test14	Test15	Test16	Test17	Test18	Test1
(AEL, 1, 1)	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
(AEL, 10, 1)	1	1	0	0	0	0	1	0	1	0	1	1	1	1	1	1	1	1	1
(AEL, 11, 1)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
(AEL, 12, 1)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
(AEL, 13, 1)	1	1	0	0	0	0	1	0	1	0	1	1	1	1	1	1	1	1	1
(AEL, 14, 1)	1	1	0	0	0	0	1	0	1	0	1	1	1	1	1	1	1	1	1
(AEL, 15, 1)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
(AEL, 16, 1)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
(AEL, 17, 1)	1	1	0	0	0	0	1	0	1	0	1	1	1	1	1	1	1	1	1
(AEL, 18, 1)	1	1	0	0	0	0	1	0	1	0	1	1	1	1	1	1	1	1	1
Mutants	Test1	Test2	Test3	Test4	Test5	Test6	Test7	Test8	Test9	Test10	Test11	Test12	Test13	Test14	Test15	Test16	Test17	Test18	Test1

Showing 1 to 10 of 298 entries Previous 1 2 3 4 5 ... 30 Next

Close Comparison results

Figura 6.14: Matriz de ejecución.

Capítulo 7

Conclusiones

La realización de este proyecto fin de carrera me ha resultado una experiencia muy grata, puesto que al haber desarrollado el proyecto en colaboración con el grupo de investigación UCASE me ha permitido conocer de primera mano como trabaja el grupo de investigación internamente con respecto a la materia tratada en este proyecto. No obstante, en el desarrollo de este proyecto también ha sido necesario un gran esfuerzo y dedicación, ya que ha supuesto un duro trabajo durante un largo período de tiempo.

Así mismo, se ha conseguido lograr el cumplimiento de todos los objetivos marcados en este proyecto. En este sentido, se ha realizado de manera exitosa la reingeniería del sistema de generación de casos de prueba para WS-BPEL 2.0 basado en un algoritmo genético, liberándolo de los problemas iniciales con los que partía e implementando correctamente el algoritmo diseñado por los autores del grupo de investigación UCASE. De la misma manera, se ha mejorado y aumentado la funcionalidad del sistema de acuerdo a las necesidades observadas mediante la realización de experimentos sobre composiciones WS-BPEL, de tal modo se ha incrementado el número de operadores de cruce y mutación de los que dispone el sistema, proveyendo un amplio abanico de operadores genéticos con una heterogeneidad significativa y bastante útil para realizar estudios de investigación. También se ha desarrollado la generación de informes en formato HTML 5 que contiene las estadísticas y resultados de cada ejecución del algoritmo. Además, gracias a la reingeniería del sistema se ha conseguido obtener resultados de investigación significativos, con los que se ha logrado la publicación de un artículo en las XX Jornadas de Ingeniería del Software y Bases de Datos, llamado «Análisis y determinación del impacto del operador de mutación en la generación genética de casos de prueba para WS-BPEL».

En dicho artículo se presentan los estudios llevados a cabo sobre el comportamiento del generador de casos de prueba basado en un algoritmo genético para matar el mayor número posible de mutantes de una composición WS-BPEL. Para ello se han realizado experimentos con tres operadores de mutación distintos aplicados a la composición *Triangle*, facilitada por el grupo de investigación del Prof. Franz Wotawa de la Technische Universität Graz, que ha sido adaptada para su correcto uso. Estos experimentos se han realizado sobre tres rangos distintos para la longitud que puede tomar los lados del triángulo, con siembra automática y sin ella, para analizar el comportamiento de cada uno de los distintos operadores de mutación aplicados y observar si mejoran los resultados obtenidos mediante la generación aleatoria de casos de prueba.

Mi colaboración en la realización del artículo presentado a JSIBD ha sido una experiencia bastante enriquecedora, puesto que es la primera vez que participo en el desarrollo de un artículo de investigación. Además, me ha permitido aprender de los debates creados en el grupo de investigación tanto para la realización del artículo como para el desarrollo del proyecto, así como también he podido conocer todo el trabajo que hay detrás de un artículo de este tipo. Ha supuesto una ardua tarea, puesto que se ha tenido que realizar una gran cantidad de experimentos.

Cabe destacar, que el desarrollo de este proyecto me ha servido como aprendizaje de muchos conceptos nuevos. En este sentido, es la primera vez que estudio algo relacionado con los algoritmos gené-

ticos, algo que me ha resultado muy interesante. También me ha servido para desenvolverme mejor con el lenguaje de programación Java y descubrir herramientas bastante útiles como C3, empleado para el desarrollo de las gráficas mostradas en los informes generados con formato HTML 5. Además, me ha permitido practicar con otras tecnologías y herramientas: JavaScript se ha empleado para la generación del informe HMTL 5. Velocity se ha utilizado como plantilla para la generación del informe HTML 5 y como plantilla para la generación automatizada de casos de pruebas. BPELUnit, que es el marco de pruebas unitarias para probar composiciones WS-BPEL. JUnit, empleado para las pruebas unitarias de Java. *Mockito* ha sido utilizado para la realización de pruebas unitarias estructurales al código Java. Así como el ya comentado HTML 5.

También se han utilizado otras herramientas para la generación de esta memoria, como el editor gráfico *yEd*, con el que se han realizado los gráficos contenidos en esta memoria. Otra de las herramientas utilizadas ha sido *GanttProject*, con la que se ha realizado el diagrama de Gantt correspondiente. Además, esta memoria se ha desarrollado mediante el empleo del lenguaje LaTeX, que resulta de gran utilidad para escribir documentos de gran tamaño con relativa comodidad.

Como posible trabajo futuro se podría plantear la realización de experimentos sobre más composiciones WS-BPEL, con los que poder estudiar la repercusión de otros aspectos importantes del algoritmo genético, como puede ser la combinación de diversos operadores de mutación o el impacto del comportamiento de los distintos operadores de cruce, y tratar de mejorar esos aspectos de acuerdo a los resultados obtenidos según las decisiones que se tomen en el grupo de investigación UCASE.

Apéndice A

Manual de usuario

En este manual se detalla el proceso a seguir para obtener y poder emplear correctamente la herramienta.

A.1. Descarga y uso de la herramienta

Podemos descargar el proyecto y todos los proyectos de los que depende de la forja del grupo UCASE: `https://neptuno.uca.es/redmine/projects/sources-fm/repository/show/branches/ode-mubpel/src`. Se puede hacer de manera sencilla mediante el empleo de *Subversion*, a través de la orden:

```
svn co https://neptuno.uca.es/svn/sources-fm/branches/ode-mubpel/src/
```

Una vez tengamos en un directorio todos los proyectos descargados, debemos compilar el proyecto y todos de los que depende. Se puede realizar fácilmente mediante la orden:

```
mvn -am -pl gamerhom-ggen compile
```

Para realizar la instalación de la herramienta, se puede emplear la siguiente orden:

```
mvn -am -pl gamerahom-ggen -DskipTests install
```

La orden anterior creará un fichero *zip* en el subdirectorio `gamerahom-ggen/target`. Es necesario descomprimir ese fichero, que creará un directorio con con todos los ficheros necesarios y el *script* para ejecutar la herramienta.

En la orden para ejecutar la herramienta, debemos indicar donde se encuentra el fichero *vm* que contiene la población inicial y un fichero de configuración *yaml*.

```
./gamerahom-ggen --initial data.vm configuration.yaml
```

A.2. Obtención de los ficheros necesarios para la ejecución

A.2.1. Fichero con la población inicial

Si disponemos de una fichero de caso de pruebas *bpts* basado en plantillas y el correspondiente fichero *spec* que define la estructura que deben seguir los datos a emplear por los casos de prueba,

podemos obtener el fichero *vm* que defina una determinada población constituida por un conjunto de casos de prueba mediante la herramienta TestGenerator:

```
testgenerator -n 30 data.spec > data.vm
```

Siendo n el número de casos de prueba debe contener el conjunto.

En el caso de que se necesite obtener previamente el fichero *spec* con el formato de los datos y el fichero de casos de pruebas *bpts* basado en plantillas, se puede emplear la herramienta *bpts-generator*:

```
bpts-generator Triangle.bpel > testSuite.bpts
```

Ambas herramientas se pueden obtener mediante la descarga y ejecución del *script* disponible en la forja del grupo de investigación UCASE: `https://neptuno.uca.es/redmine/projects/sources-fm/repository/changes/trunk/scripts/install.sh`.

```
./install gamera
```

A.2.2. Fichero de configuración YAML

En el fichero de configuración *yaml* se puede determinar las distintas características deseadas para una determinada ejecución, tomando como ejemplo el fichero de configuración mostrado en el listado 4.2.

YAML es un formato con el que podemos codificar objetos de una manera sencilla y que favorece la legibilidad de los datos. Permite una fácil representación de los datos mediante combinaciones de *Maps*, listas y valores simples.

Se trata de un formato fácilmente legible puesto que emplea una notación basada en la indentación mediante el uso de espacios en blanco.

Para indicar cada uno de los elementos de una lista se emplea un guión seguido de un espacio en blanco, de manera que cada elemento se indique en una nueva línea.

```
1 loggers:
2   - logger1
3   - logger2
4   - logger3
```

También se puede definir una lista mediante el empleo de corchetes, separando cada uno de sus elementos mediante comas.

```
1 loggers: [logger1, logger2, logger3]
```

Para definir un *Map* se emplean dos puntos seguidos de un espacio en blanco, de manera que cada entrada del *Map* se indique en una nueva línea.

```
1 ops:
2   Int : op1
3   Float : op2
4   String : op3
```

También se puede definir un *Map* mediante el empleo de llaves, separando cada uno de sus entradas mediante comas.

```
ops: {Int : op1, Float : op2, String : op3}
```

Además, mediante el empleo de «?» podemos indicar una clave compleja de un *Map*.

```

? - op1
  - op2
:
  - pm
```

Así mismo, los comentarios se pueden indicar mediante «#».

Para aprender mejor sobre el uso de YAML, se pueden observar ejemplos como los que se muestran en `http://www.yaml.org/spec/1.2/spec.html#Preview`.

A.3. Combinación de operadores de mutación

Podemos especificar varios operadores de mutación a utilizar en una determinada ejecución. Además, gracias al diseño del patrón *Composite* podemos combinar varios operadores de mutación y establecer la aplicación de ellos según el tipo de la variable a mutar o la posición que ocupa.

Tenemos la posibilidad de especificar en el fichero de configuración YAML los operadores de mutación a emplear de las siguientes formas:

- Utilización de un solo operador de mutación.

```
mutationOperators:
  !!gamera.ggen.genetic.mutation.BinaryMutationOperator {mutationRange
     : 20} : {probability: 0.1}
```

Hay determinados operadores de mutación que necesitan que se especifiquen ciertos parámetros. Para el operador *Creep Mutation* y todos los que lo extienden hay que determinar el valor que debe tomar el parámetro *mutationRange*, que modula el rango del cambio que puede sufrir un determinado valor, y opcionalmente el parámetro *compositeMutOp*, que especifica el operador a emplear par mutar los elementos contenidos en listas y tuplas.

Para el operador de mutación de listas con probabilidad personalizada, a parte de los dos parámetros comentados anteriormente, es necesario especificar los parámetros de probabilidad de eliminación (*probabilityElimination*), probabilidad de inserción (*probabilityInsertion*) y opcionalmente la probabilidad de mutación de elementos (*probabilityMutationElement*). El valor de esta última probabilidad solo se tendría en cuenta si las probabilidades de eliminación e inserción son cero, en este caso tomaría por defecto el valor 0.999 si no se ha especificado valor alguno para dicha probabilidad. Cabe recordar que la probabilidad de mutación de elementos se calcula: 1 - *probabilityElimination* - *probabilityInsertion*.

- Utilización de varios operadores de mutación.

```
mutationOperators:
  !!gamera.ggen.genetic.mutation.BinaryMutationOperator {mutationRange
     : 20} : {probability: 0.1}
  !!gamera.ggen.genetic.mutation.TriangleMutationOperator {} : {
     probability: 0.1}
```

- Combinación de operadores de mutación por tipos. Podemos especificar que operador de mutación queremos aplicar para cada distinto tipo de dato. También sirve para los tipos de datos definidos por el usuario.

```
mutationOperators:
  ? !!gamera.ggen.genetic.mutation.CompositeByTypeMutationOperator
    ops:
      Int : !!gamera.ggen.genetic.mutation.BinaryMutationOperator {
          mutationRange: 20}
      Float : !!gamera.ggen.genetic.mutation.
          GrayBinaryMutationOperator {mutationRange: 20}
      String : !!gamera.ggen.genetic.mutation.MutationOperator {
          mutationRange: 20}
      List : !!gamera.ggen.genetic.mutation.
          CustomProbabilityListMutationOperator
        mutationRange: 20
        probabilityElimination: 0.2
        probabilityInsertion: 0.2
      Tuple: !!gamera.ggen.genetic.mutation.MutationOperator {
          mutationRange : 20}
  : {probability: 0.1}
```

En el caso de que se desee utilizar el mismo operador para distintos tipos, también se puede emplear el siguiente formato para la especificación de los distintos operadores de mutación a aplicar. De manera que los operadores que se repitan para distintos tipos, se puedan especificar mediante una referencia.

```
mutationOperators:
  ? !!gamera.ggen.genetic.mutation.CompositeByTypeMutationOperator
    ops:
      Int : &x !!gamera.ggen.genetic.mutation.BinaryMutationOperator {
          mutationRange: 20}
      Float : *x
      String : &y!!gamera.ggen.genetic.mutation.MutationOperator {
          mutationRange: 20}
      List : !!gamera.ggen.genetic.mutation.
          CustomProbabilityListMutationOperator
        mutationRange: 20
        probabilityElimination: 0.2
        probabilityInsertion: 0.2
      Tuple: *y
  : {probability: 0.1}
```

- Combinación de operadores de mutación por posiciones. Podemos especificar que operador de mutación queremos aplicar para cada componente según la posición que ocupan en un individuo de la población.

```
mutationOperators:
  ? !!gamera.ggen.genetic.mutation.CompositeByPositionMutationOperator
    ops:
      - !!gamera.ggen.genetic.mutation.BinaryMutationOperator {
          mutationRange: 20}
```

```
5         - !!gamera.ggen.genetic.mutation.GrayBinaryMutationOperator {
              mutationRange: 20}
6         - !!gamera.ggen.genetic.mutation.MutationOperator {mutationRange
              : 20}
7         - !!gamera.ggen.genetic.mutation.
              CustomProbabilityListMutationOperator
8           mutationRange: 20
9           probabilityElimination: 0.2
10          probabilityInsertion: 0.2
11        - !!gamera.ggen.genetic.mutation.UniformMutationOperator {
              mutationRange: 20}
12    : {probability: 0.1}
```

De la misma manera, en el caso de que se desee utilizar el mismo operador para distintas posiciones, también se puede emplear el siguiente formato para la especificación de los distintos operadores de mutación a aplicar. De modo que los operadores que se repitan para distintas posiciones, se puedan especificar mediante una referencia.

```
1   mutationOperators:
2     ? !!gamera.ggen.genetic.mutation.CompositeByPositionMutationOperator
3       ops:
4         - &x !!gamera.ggen.genetic.mutation.BinaryMutationOperator {
              mutationRange: 20}
5         - &y !!gamera.ggen.genetic.mutation.GrayBinaryMutationOperator {
              mutationRange: 20}
6         - *x
7         - *y
8         - !!gamera.ggen.genetic.mutation.MutationOperator {mutationRange
              : 20}
9     : {probability: 0.1}
```

Por otro lado, para la mutación de listas y tuplas también podemos especificar el operador de mutación a emplear para mutar los elementos contenidos por éstas. Si no se indica ninguno, se empleará el operador establecido por defecto, que es el operador de mutación *Creep Mutation*. Se puede especificar de la siguiente manera:

```
1   mutationOperators:
2     ? !!gamera.ggen.genetic.mutation.
        CustomProbabilityListMutationOperator
3       mutationRange: 20
4       probabilityElimination: 0.2
5       probabilityInsertion: 0.2
6       compositeMutOp: !!gamera.ggen.genetic.mutation.
          BinaryMutationOperator {mutationRange: 20}
7     : {probability: 0.1}
```

A.4. Empleo de la herramienta *MuBPEL*

La herramienta *MuBPEL* nos facilita el trabajo a realizar con los conjuntos de casos de prueba aleatorios que se generan para decidir la población inicial que va a tomar como entrada el algoritmo genético.

Esta herramienta nos proporciona distintas funciones:

- **Listar operadores aplicables a una composición:** Podemos obtener una lista con todos los operadores que se pueden aplicar a una determinada composición. Para ello, tenemos que escribir la siguiente orden:

```
mubpel analyze fichero.bpel
```

  Con dicha orden, se genera una lista donde se muestra el nombre de cada operador seguido de dos números: el primero representa el número de localizaciones en las que se puede aplicar dicho operador en la composición y el segundo representa el atributo de cada operador, es decir, el número de formas distintas en las que se puede aplicar en cada localización.

- **Generar mutantes de una composición:** Podemos generar todos los mutantes posibles a partir una determinada composición. Lo podemos realizar escribiendo la orden:

```
mubpel applyall fichero.bpel
```

  Esta herramienta además nos permite generar mutantes determinados indicando el operador a aplicar, la localización en la composición donde aplicar el operador y el atributo que debe tomar.

```
mubpel apply fichero.bpel operator operandIndex attribute > mutante.bpel
```

  Donde *operator*, *operandIndex* y *attribute* son números que representan, respectivamente, al operador a aplicar, la localización en la composición donde aplicar el operador y un atributo específico.

- **Ejecución de una composición:** Para ejecutar una composición frente a un conjunto de casos de prueba debemos emplear la siguiente orden:

```
mubpel run fichero.bpts fichero.bpel > salida.xml
```

  Con dicha orden podemos obtener los resultados obtenidos por la composición al ejecutarla contra el fichero de casos de pruebas *bpts*. Es conveniente volcar la salida en un archivo para poder observar fácilmente los resultados obtenidos.

- **Comparar la salida de la composición y la de los mutantes:** Podemos comparar la salida de la ejecución de la composición con la de los mutantes mediante el empleo dos órdenes:

  Con la orden *compare* podemos comparar la salida de la composición y la de los mutantes con respecto a cada uno de los casos de prueba hasta que se produzca la primera diferencia entre ellas en un determinado caso de prueba.

```
mubpel compare fichero.bpts fichero.bpel salida.xml (fichero1.bpel...|-)
```

  La salida que se obtiene es una matriz de ejecución. Consta de una fila por cada mutante con el que se realiza la comparación, dicha fila está formada por el mutante seguido de una serie de números, representando cada número al resultado obtenido para cada caso de prueba.

  - Si el número es 0, la salida de la composición y el mutante coinciden.
  - Si el número es 1, la salida de la composición y el mutante son distintas.

- Si el número es 2, se trata de un mutante inválido que no puede ser ejecutado.

Ya que se compara hasta encontrar la primera diferencia, la comparación se realizará hasta obtener el primer 1 de la fila y el resto de casos de prueba quedará a 0. Si se trata de un mutante inválido, la fila entera quedará a 2.

Además, podemos emplear la orden *compare -k* para comparar la salida de la ejecución de la composición con la de los mutantes contra todos los casos de pruebas existentes. En este caso, se realiza la comparación teniendo en cuenta todos los casos de pruebas disponibles, en lugar de detener la comparación al obtener la primera diferencia.

```
mubpel compare -k fichero.bpts fichero.bpel salida.xml (fichero1.bpel...|-)
```

- **Comparar dos salidas de ejecución de una composición:** La siguiente orden nos permite comparar dos salidas de la ejecución de una composición, realizando la comparación por cada caso de prueba:

```
mubpel compareout salida1.xml salida2.xml
```

- **Normalizar una composición:**

  Con la siguiente orden podemos normalizar una composición WS-BPEL, generando una forma canónica de tal composición:

```
mubpel normaliza fichero.bpel
```

  Resulta útil cuando se desea emplear una herramienta para poder ver las posibles diferencias entre la composición y un mutante generado a partir de ella.

- **Recuento de los mutantes que se pueden producir:** Con la siguiente orden podemos obtener el número de mutantes que se pueden producir a partir de un determinado programa original:

```
mubpel count fichero.bpel
```

Apéndice B

Manual de desarrollador

En este manual se explican los pasos a realizar para poder trabajar en el desarrollo de la herramienta.

B.1. Desarrollo con Eclipse

Para la descarga y compilación del proyecto se deben realizar los pasos indicados en la sección correspondiente del manual de usuario A.1. Una vez los hayamos realizados, debemos preparar el proyecto Eclipse [23], para ellos debemos ejecutar la siguiente orden:

```
mvn eclipse:eclipse
```

De esta manera, ya podemos importar el proyecto y los proyectos de los que depende en Eclipse.

Para importar el proyecto desde Eclipse, en primer lugar tenemos que pulsar en *File* y seleccionar *Import*. A continuación, en la ventana que se abre, debemos seleccionar la opción *Existing Projects into Workspace* dentro de la pestaña *General*. Por último, especificamos la ruta donde se encuentra el proyecto a importar y se seleccionan los proyectos que se deseen.

Por otro lado, para ejecutar las pruebas automatizadas es necesario seleccionar en *Project Explorer* el test a ejecutar, pulsar con el botón derecho y seleccionar *Run As* $\rightarrow$ *JUnit Test*. Otra forma de ejecutar un test es emplear la combinación de teclas *Shift+Alt+X T*.

B.2. Generación de informe HTML 5 mediante Velocity

Podemos emplear el motor de plantillas Velocity para generar código HTML 5 que sigue una estructura marcada por una plantilla Velocity, siendo la información contenida los resultados obtenidos mediante la aplicación Java. En este sentido, Velocity facilita la representación de los datos separando la funcionalidad de la aplicación Java y la plantilla Velocity, lo que mejora la mantenibilidad y actualización de la representación de los datos.

De esta manera, por un lado es necesario diseñar una plantilla Velocity con la que fijar la estructura a seguir para representar los datos. A modo de ejemplo, podemos ver un fragmento de la plantilla Velocity diseñada en el siguiente listado:

Listado B.1: Fragmento de plantilla Velocity

```
1 ........
2 <table id="generatedIndividuals$generationBlockCount" class="display
     resultsTable" cellspacing="0">
3           ........
4           <tbody>
```

```
                #foreach($individualGenerated in $generatedIndividuals.get(
                    $generationBlockCount))
                        <tr>
                                #foreach($generatedValue in
                                    $individualGenerated)
                                        <td>$generatedValue</td>
                                #end
                        </tr>
                #end
        </tbody>
</table>
........
```

Como podemos ver, a través de Velocity podemos acceder a todos los métodos públicos de los objetos que contienen los datos a representar. En el caso concreto del ejemplo mostrado, utilizamos un *Map* que contiene todos los individuos generados en cada generación.

Asimismo, debemos crear el *logger* en Java para obtener la información que será representada siguiendo la estructura marcada en la plantilla Velocity. Para poder hacer uso en Java del motor de plantillas Velocity, debemos realizar las siguientes acciones:

- Crear un objeto *VelocityEngine* e inicializarlo.

```
VelocityEngine ve = new VelocityEngine();
ve.init();
```

- Indicar la plantilla a emplear para generar el código html.

```
Template t = ve.getTemplate("vm/statistic-ggen.vm");
```

- Crear un contexto que permite contener la información generada para que se pueda acceder a ella a través de la plantilla Velocity.

```
VelocityContext context = new VelocityContext();
```

- Almacenar en el contexto la información necesaria.

```
private final Map<Integer,List<List<String>>>
    generatedIndividualsByGeneration = new HashMap<Integer,List<List<
    String>>>();
........
context.put("generatedIndividuals", generatedIndividualsByGeneration);
........
```

- Generar el fichero correspondiente con la información generada representada a través de la plantilla.

```
private PrintStream ps = null;
........
if(ps == null) {
        try {
```

```
5                   ps = new PrintStream(new File(file));
6           } catch (Exception e) {
7                   LOGGER.error("File not found", e);
8           }
9   }
10  ........
11  StringWriter writer = new StringWriter();
12  t.merge(context, writer);
13  ps.println(writer.toString());
```

De este modo, hemos podido generar código HMTL 5 de una manera sencilla, donde se consigue mostrar de forma clara los resultados obtenidos por el algoritmo genético en una determinada ejecución. Además, gracias al empleo de la plantilla Velocity se realiza la representación de los datos de una manera independiente al código Java, por lo que la plantilla se puede modificar sin necesidad de alterar el *logger* realizado en Java.

B.3. Herramientas

B.3.1. Eclipse

Eclipse ha sido la herramienta empleada para el desarrollo del código Java del proyecto. Se trata de un entorno de desarrollo integrado multiplataforma que permite el desarrollo de aplicaciones bajo prácticamente cualquier lenguaje de programación, gracias al sistema de plug-in que posee para personalizar el entorno de trabajo.

Eclipse posee una interfaz gráfica bastante intuitiva que facilita el desarrollo software. Asimismo, posee funcionalidades muy potentes y útiles que hacen que el desarrollo del código sea mucho más cómodo.

En la figura B.1 podemos observar la interfaz del entorno de desarrollo Eclipse.

B.3.2. yEd

yEd [24] es un potente editor de gráficos con el que podemos generar diagramas de gran calidad de una forma sencilla, rápida y eficaz. Mediante este editor se puede realizar una gran variedad de tipos de diagramas, como: diagramas UML, diagramas de flujo, diagramas de entidad-relación, diagramas BPMN, etc.

Esta herramienta presenta una interfaz muy intuitiva y amigable, ofrece una paleta en la que podemos seleccionar los objetos a utilizar según según el tipo de diagrama que se desee realizar. Para realizar un diagrama, simplemente se arrastran los objetos desde la paleta al documento en el que se esté trabajando.

En la figura B.2 podemos observar la interfaz que presenta el editor de gráficos *yEd*.

Asimismo, ofrece herramientas muy potentes como la creación automática de árboles y redes, así como la manipulación y transformación geométrica de estos objetos de forma automática. Entre otras cosas útiles, también posibilita el empleo de una cuadrícula a la que se pueden ajustar los distintos objetos.

yEd guarda los diagramas creados con un formato basado en XML, *GraphML*. Además, permite la exportación de los diagramas creados a diferentes formatos, como pdf, png, svg, swf, html, etc.

B.3.3. MuBPEL

MuBPEL es una herramienta desarrollada por el grupo de investigación UCASE. Esta herramienta se emplea para la ejecución de las composiciones WS-BPEL frente a un conjunto de casos de prueba.

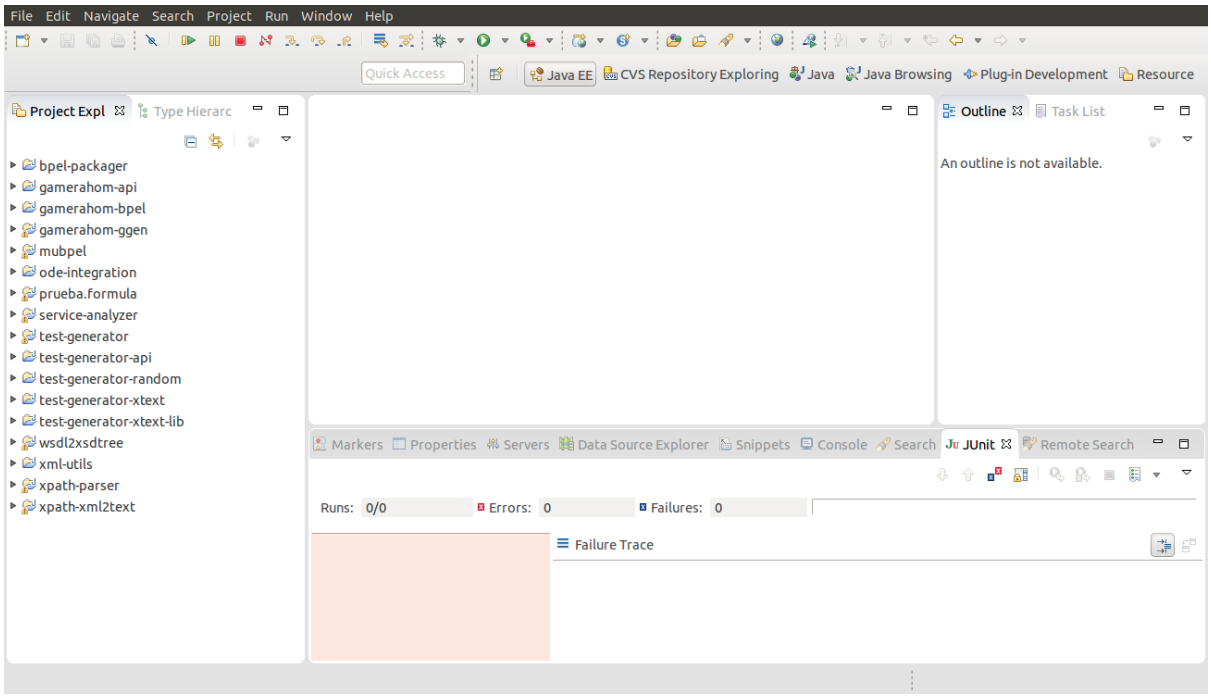


Figura B.1: Interfaz de Eclipse.

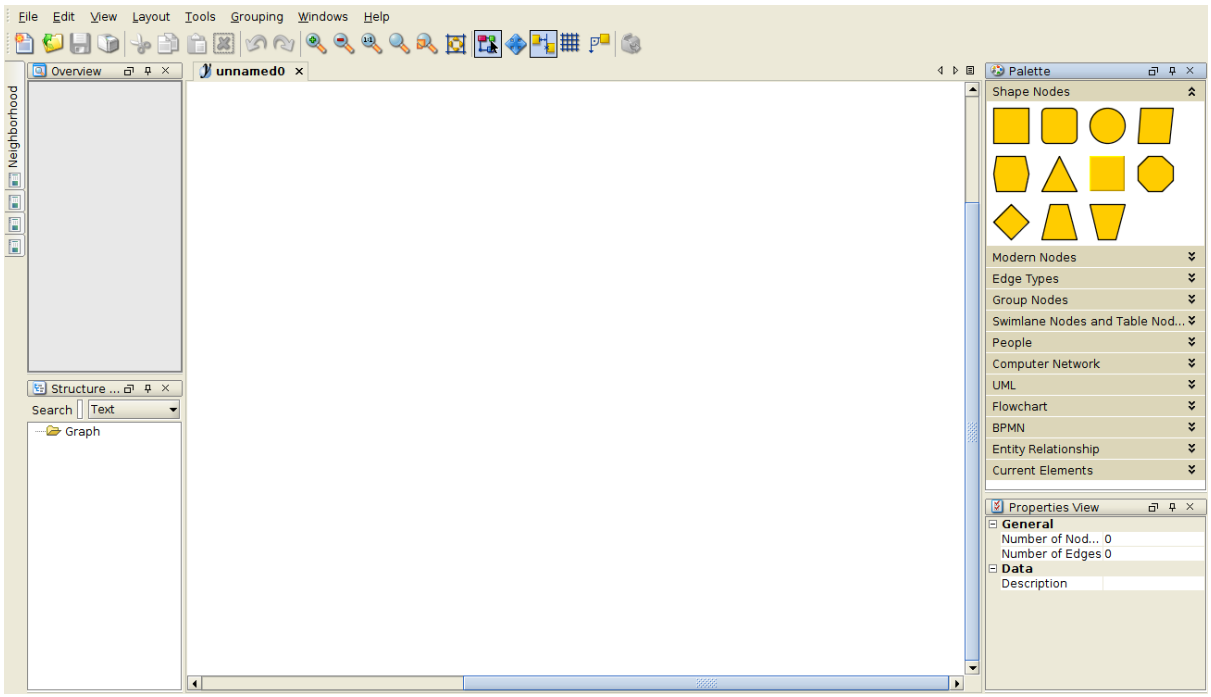


Figura B.2: Interfaz del editor de gráficos *yEd*.

Asimismo, también se utiliza para generar todos los mutantes a partir de una composición original y realizar la comparación entre la composición y los mutantes obtenidos a partir de ella. Entre otras cosas, también permite analizar cuantos mutantes se pueden obtener de una composición WS-BPEL.

Todas las funciones de MuBPEL son las siguientes. Las cuales podemos obtener al escribir en la terminal:

```
mubpel -h
```

```
Available subcommands:
 * analyze bpel
 * apply bpel operator operandIndex attribute
 * applyall bpel
 * compare bpts bpel xml (bpel1...|-)
 * comparefull bpts bpel xml (bpel1...|-)
 * compareout xml xml1...
 * normalize bpel
 * run bpts (bpel1...|-)
```

B.3.4. Apache ODE

Apache ODE [25] es un motor que permite la ejecución de procesos de negocio WS-BPEL. Facilita la comunicación con servicios webs a través de un intercambio de mensajes, aunque también permite sustituir los servicios reales por servicios emulados. Existen diversos motivos por los que se puede desear hacer este tipo de sustitución:

- Los servicios reales no se encuentran disponibles disponibles, o no se desea interactuar con ellos por alguna razón.
- Los servicios reales bloquean recursos, no son gratis, etc.
- Se desea especificar cuáles van a ser las respuestas o peticiones a los distintos servicios para poder controlar en todo momento el entorno de prueba.

B.3.5. TkDiff

TkDiff [26] es una herramienta gráfica con la que podemos observar las diferencias existentes entre dos ficheros. Esta herramienta muestra, a pantalla compartida, los dos ficheros marcando línea a línea las diferencias encontradas entre ambos.

De este modo, podemos utilizar TkDiff para comparar un fichero original *bpel* con un mutante generado por un determinado operador de mutación, de esta manera podemos observar qué parte del código se ha modificado y estudiar el resultado obtenido al ejecutarlo contra el conjunto de casos de prueba utilizado.

Para realizar una comparación entre dos ficheros podemos escribir directamente la siguiente orden en una terminal.

```
tkdiff fichero1.bpel fichero2.bpel
```

En la figura B.3 podemos observar la interfaz de TkDiff.

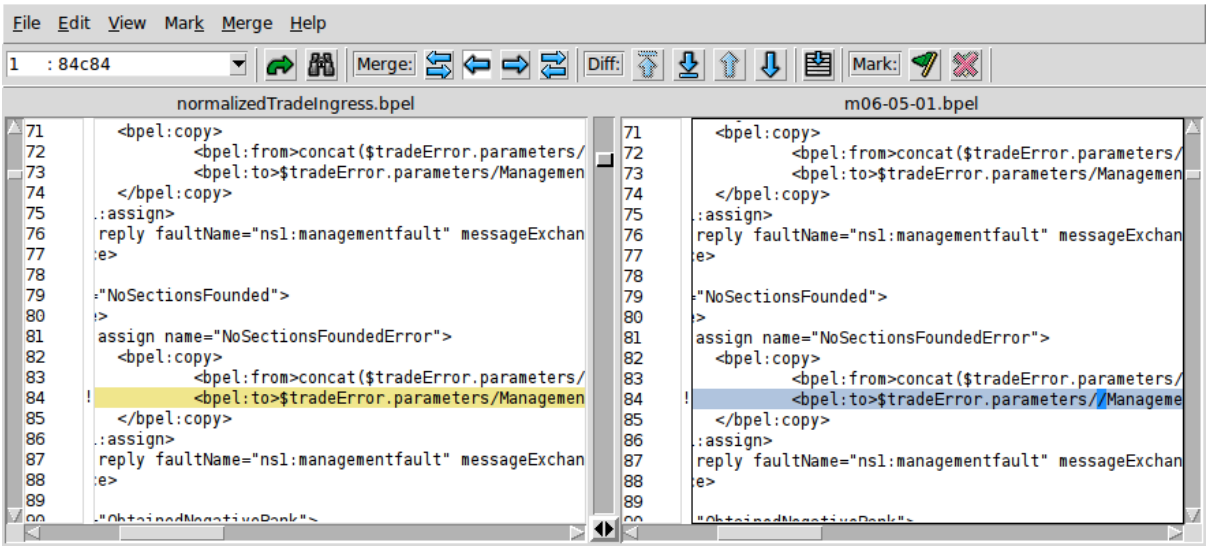


Figura B.3: Herramienta TkDiff

B.3.6. XMLEye

XMLEye [27] es un visor genérico de documentos basados en XML. No está ligado a ningún formato en particular, sino que puede abrir casi cualquier tipo de documento, que debe ser transformado previamente a XML para poder visualizarse.

Esta herramienta es muy útil para visualizar las salidas que producen las composiciones al ejecutarlas contra los casos de prueba y comprobar que todos se han ejecutado correctamente.

Para visualizar un documento con XMLEye podemos escribir directamente la siguiente orden en una terminal.

```
xmleye salida.xml
```

En la figura B.4 podemos observar la interfaz de XMLEye.

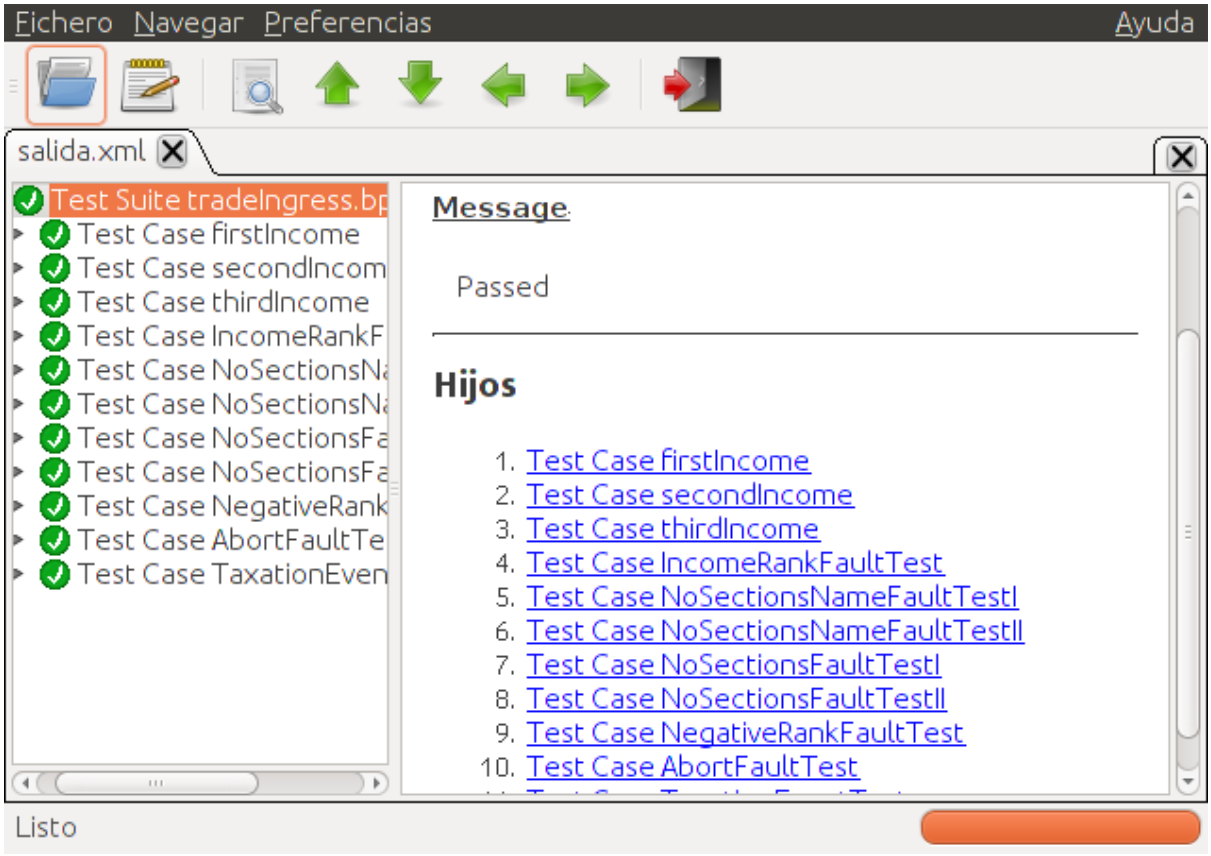


Figura B.4: Herramienta XMLEye

Bibliografía

[1] OASIS. Web Services Business Process Execution Language 2.0, 2007.
URL http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

[2] M. Mitchell. *An Introduction to Genetic Algorithms*. The MIT Press, 1998.

[3] A. Galán-Piñero, A. García-Domínguez y J. J. Domínguez-Jiménez. *Generador de casos de prueba genético*. Proyecto fin de carrera, Universidad de Cádiz, Cádiz, España, 2012.
URL http://rodin.uca.es/xmlui/bitstream/handle/10498/14877/memoria.pdf?sequence=1

[4] A. J. Offutt y R. H. Untch. *Mutation Testing for the New Century*, capítulo Mutation 2000: Uniting the Orthogonal, páginas 34–44. Kluwer Academic Publishers, 2001.

[5] BPELUnit. The Open Source Unit Testing Framework for BPEL, Septiembre 2011.
URL http://bpelunit.net/

[6] Apache. Velocity, Noviembre 2010.
URL http://velocity.apache.org/

[7] Sun Microsystems (Oracle Corporation). Java, 2014.
URL https://www.java.com/es/

[8] K. Beck, E. Gamma y D. Saff. JUnit, 2012.
URL junit.org

[9] LaTeX3 Project. LaTeX, 2011.
URL http://www.latex-project.org/

[10] A. Estero-Botaro. *Aplicación de la prueba de mutaciones a composiciones de servicios web en WS-BPEL para la generación de casos de prueba de calidad*. Tesis, Universidad de Cádiz, Cádiz, España, 2013.
URL http://hdl.handle.net/10498/17562

[11] Apache Software Foundation. Subversion, 2015.
URL https://subversion.apache.org/

[12] M. Ángel Pérez Montero. TestSpec. En *Generador de casos de prueba aleatorio basado en especificaciones abstractas*. 2012.

[13] E. Gamma, R. Helm, R. Johnson y J. Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[14] World Wide Web Consortium. HTML5, Octubre 2014.
URL http://www.w3.org/TR/html5/

[15] M. Ángel Pérez Montero. *Generador de casos de prueba basado en estrategias personalizables*. Proyecto fin de carrera, Universidad de Cádiz, Cádiz, España, 2013.
URL http://rodin.uca.es/xmlui/bitstream/handle/10498/15486/memoria.pdf?sequence=1

[16] A. Estero-Botaro, F. Palomo-Lozano y I. Medina-Bulo. Mutation operators for WS-BPEL 2.0. En *ICSSEA 2008, 21th International Conference on Software & Systems Engineering and their applications*. 2008.

[17] A. García-Domínguez, A. Estero-Botaro, F. Palomo-Lozano y I. Medina-Bulo. MuBPEL: una herramienta de mutación firme para WS-BPEL 2.0. En *Actas de las XVI Jornadas de Ingeniería del Software y Bases de Datos*. 2012.

[18] O. Ben-Kiki, C. Evans y I. döt Net. Yaml, 2009.
URL http://www.yaml.org/spec/1.2/spec.html

[19] S. Faber. Mockito, 2008.
URL http://mockito.org/

[20] SpryMedia Ltd. Datatables, 2014.
URL http://www.datatables.net

[21] M. Tanaka. C3, 2014.
URL http://c3js.org/

[22] M. Bostock. D3, 2015.
URL http://d3js.org/

[23] Eclipse Foundation. Eclipse, 2015.
URL https://eclipse.org/

[24] yWorks GmbH. yEd, 2015.
URL http://www.yworks.com/en/products/yfiles/yed/

[25] Apache Software Foundation. Apache ODE, 2013.
URL http://ode.apache.org/

[26] sourceforge.net. TkDiff, Mayo 2012.
URL http://sourceforge.net/projects/tkdiff/

[27] A. G. Domínguez. XMLEye, Septiembre 2011.
URL https://neptuno.uca.es/redmine/projects/sources-fm/wiki/XMLEye